Python Library Reference

Release 2.1

Guido van Rossum
Fred L. Drake, Jr., editor

April 15, 2001

PythonLabs
E-mail: python-docs@python.org
BEOPEN.COM TERMS AND CONDITIONS FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI OPEN SOURCE GPL-COMPATIBLE LICENSE AGREEMENT

Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.

CWI PERMISSIONS STATEMENT AND DISCLAIMER

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Abstract

Python is an extensible, interpreted, object-oriented programming language. It supports a wide range of applications, from simple text processing scripts to interactive WWW browsers.

While the Python Reference Manual describes the exact syntax and semantics of the language, it does not describe the standard library that is distributed with the language, and which greatly enhances its immediate usability. This library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs.

This library reference manual documents Python’s standard library, as well as many optional library modules (which may or may not be available, depending on whether the underlying platform supports them and on the configuration choices made at compile time). It also documents the standard types of the language and its built-in functions and exceptions, many of which are not or incompletely documented in the Reference Manual.

This manual assumes basic knowledge about the Python language. For an informal introduction to Python, see the Python Tutorial; the Python Reference Manual remains the highest authority on syntactic and semantic questions. Finally, the manual entitled Extending and Embedding the Python Interpreter describes how to add new extensions to Python and how to embed it in other applications.
# CONTENTS

## 1 Introduction

1

## 2 Built-in Types, Exceptions and Functions

1.1 Built-in Types .................................................. 3
1.2 Built-in Exceptions ........................................... 16
1.3 Built-in Functions .............................................. 20

## 3 Python Runtime Services

3.1 sys — System-specific parameters and functions .................... 29
3.2 gc — Garbage Collector interface ................................ 34
3.3 weakref — Weak references ..................................... 35
3.4 fpectl — Floating point exception control .......................... 39
3.5 atexit — Exit handlers ......................................... 40
3.6 types — Names for all built-in types ............................. 41
3.7 UserDict — Class wrapper for dictionary objects .................. 43
3.8 UserList — Class wrapper for list objects .......................... 43
3.9 UserString — Class wrapper for string objects ...................... 44
3.10 operator — Standard operators as functions ....................... 44
3.11 inspect — Inspect live objects ................................. 48
3.12 traceback — Print or retrieve a stack traceback .................... 52
3.13 linecache — Random access to text lines ........................ 53
3.14 pickle — Python object serialization ............................ 54
3.15 cPickle — Alternate implementation of pickle .................... 59
3.16 copy_reg — Register pickle support functions ...................... 59
3.17 shelve — Python object persistence ............................. 59
3.18 copy — Shallow and deep copy operations ........................ 60
3.19 marshal — Alternate Python object serialization .................. 61
3.20 warnings — Warning control .................................... 62
3.21 imp — Access the import internals ................................ 64
3.22 code — Interpreter base classes .................................. 67
3.23 codeop — Compile Python code ................................... 69
3.24 pprint — Data pretty printer .................................... 69
3.25 repr — Alternate repr() implementation .......................... 71
3.26 new — Creation of runtime internal objects ...................... 73
3.27 site — Site-specific configuration hook .......................... 73
3.28 user — User-specific configuration hook .......................... 74
3.29 __builtins__ — Built-in functions ................................ 75
3.30 __main__ — Top-level script environment .......................... 75
# 4 String Services

4.1 string — Common string operations .......................................................... 77
4.2 re — Regular expression operations ............................................................ 80
4.3 struct — Interpret strings as packed binary data ......................................... 88
4.4 difflib — Helpers for computing deltas ...................................................... 90
4.5 fpformat — Floating point conversions ...................................................... 94
4.6 StringIO — Read and write strings as files ............................................... 95
4.7 cStringIO — Faster version of StringIO .................................................. 95
4.8 codecs — Codec registry and base classes ................................................. 96
4.9 unicodedata — Unicode Database ............................................................. 100

# 5 Miscellaneous Services

5.1 doctest — Test docstrings represent reality .............................................. 103
5.2 unittest — Unit testing framework ........................................................... 110
5.3 math — Mathematical functions ................................................................. 119
5.4 cmath — Mathematical functions for complex numbers ............................ 121
5.5 random — Generate pseudo-random numbers ........................................... 122
5.6 wrandom — Pseudo-random number generator ......................................... 125
5.7 bisect — Array bisection algorithm .......................................................... 125
5.8 array — Efficient arrays of numeric values ................................................ 126
5.9 ConfigParser — Configuration file parser ................................................ 129
5.10 fileinput — Iterate over lines from multiple input streams ....................... 131
5.11 xreadlines — Efficient iteration over a file ............................................. 132
5.12 calendar — General calendar-related functions ........................................ 133
5.13 cmd — Support for line-oriented command interpreters ............................ 134
5.14 shlex — Simple lexical analysis ............................................................... 136

# 6 Generic Operating System Services

6.1 os — Miscellaneous OS interfaces ............................................................ 139
6.2 os.path — Common pathname manipulations ............................................ 150
6.3 dircache — Cached directory listings ....................................................... 152
6.4 stat — Interpreting stat() results ............................................................. 153
6.5 statcache — An optimization of os.stat() ............................................... 155
6.6 statvfs — Constants used with os.statvfs() ............................................ 156
6.7 filecmp — File and Directory Comparisons ............................................. 156
6.8 popen2 — Subprocesses with accessible I/O streams ................................. 158
6.9 time — Time access and conversions ....................................................... 159
6.10 sched — Event scheduler ....................................................................... 162
6.11 mutex — Mutual exclusion support ........................................................ 164
6.12 getpass — Portable password input ........................................................ 164
6.13 curses — Terminal handling for character-cell displays ............................. 164
6.14 curses.textpad — Text input widget for curses programs ........................ 179
6.15 curses.wrapper — Terminal handler for curses programs ......................... 180
6.16 curses.ascii — Utilities for ASCII characters ......................................... 181
6.17 curses.panel — A panel stack extension for curses ................................. 183
6.18 getopt — Parser for command line options ............................................. 184
6.19 tempfile — Generate temporary file names ............................................. 186
6.20 errno — Standard errno system symbols ................................................. 186
6.21 glob — UNIX style pathname pattern expansion ..................................... 192
6.22 fnmatch — UNIX filename pattern matching ......................................... 193
6.23 shutil — High-level file operations ......................................................... 193
6.24 locale — Internationalization services .................................................... 195
6.25 gettext — Multilingual internationalization services ............................... 199
# Optional Operating System Services

1. **signal** — Set handlers for asynchronous events
2. **socket** — Low-level networking interface
3. **select** — Waiting for I/O completion
4. **thread** — Multiple threads of control
5. **threading** — Higher-level threading interface
6. **Queue** — A synchronized queue class
7. **mmap** — Memory-mapped file support
8. **anydbm** — Generic access to DBM-style databases
9. **dumbdbm** — Portable DBM implementation
10. **dbhash** — DBM-style interface to the BSD database library
11. **whichdb** — Guess which DBM module created a database
12. **bsddb** — Interface to Berkeley DB library
13. **zlib** — Compression compatible with gzip
14. **gzip** — Support for gzip files
15. **zipfile** — Work with ZIP archives
16. **readline** — GNU readline interface
17. **rlcompleter** — Completion function for GNU readline

# Unix Specific Services

8.1 **posix** — The most common POSIX system calls
8.2 **pwd** — The password database
8.3 **grp** — The group database
8.4 **crypt** — Function to check UNIX passwords
8.5 **dl** — Call C functions in shared objects
8.6 **dbm** — Simple “database” interface
8.7 **gdbm** — GNU’s reinterpretation of dbm
8.8 **termios** — POSIX style tty control
8.9 **TERMIOS** — Constants used with the termios module
8.10 **tty** — Terminal control functions
8.11 **pty** — Pseudo-terminal utilities
8.12 **fcntl** — The fcntl() and ioctl() system calls
8.13 **pipes** — Interface to shell pipelines
8.14 **posixfile** — File-like objects with locking support
8.15 **resource** — Resource usage information
8.16 **nis** — Interface to Sun’s NIS (Yellow Pages)
8.17 **syslog** — UNIX syslog library routines
8.18 **commands** — Utilities for running commands

# The Python Debugger

9.1 **Debugger Commands**
9.2 **How It Works**

# The Python Profiler

10.1 **Introduction to the profiler**
10.2 **How Is This Profiler Different From The Old Profiler?**
10.3 **Instant Users Manual**
10.4 **What Is Deterministic Profiling?**
10.5 **Reference Manual**
10.6 **Limitations**
10.7 **Calibration**
10.8 **Extensions — Deriving Better Profilers**

# Internet Protocols and Support

11.1 **webbrowser** — Convenient Web-browser controller
<table>
<thead>
<tr>
<th>Section</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.2</td>
<td>cgi</td>
<td>Common Gateway Interface support</td>
</tr>
<tr>
<td>11.3</td>
<td>urllib</td>
<td>Open arbitrary resources by URL</td>
</tr>
<tr>
<td>11.4</td>
<td>urllib2</td>
<td>Extensible library for opening URLs</td>
</tr>
<tr>
<td>11.5</td>
<td>httplib</td>
<td>HTTP protocol client</td>
</tr>
<tr>
<td>11.6</td>
<td>ftplib</td>
<td>FTP protocol client</td>
</tr>
<tr>
<td>11.7</td>
<td>gopherlib</td>
<td>Gopher protocol client</td>
</tr>
<tr>
<td>11.8</td>
<td>poplib</td>
<td>POP3 protocol client</td>
</tr>
<tr>
<td>11.9</td>
<td>imaplib</td>
<td>IMAP4 protocol client</td>
</tr>
<tr>
<td>11.10</td>
<td>nntplib</td>
<td>NNTP protocol client</td>
</tr>
<tr>
<td>11.11</td>
<td>smtplib</td>
<td>SMTP protocol client</td>
</tr>
<tr>
<td>11.12</td>
<td>telnetlib</td>
<td>Telnet client</td>
</tr>
<tr>
<td>11.13</td>
<td>urlparsing</td>
<td>Parse URLs into components</td>
</tr>
<tr>
<td>11.14</td>
<td>SocketServer</td>
<td>A framework for network servers</td>
</tr>
<tr>
<td>11.15</td>
<td>BaseHTTPServer</td>
<td>Basic HTTP server</td>
</tr>
<tr>
<td>11.16</td>
<td>SimpleHTTPServer</td>
<td>Simple HTTP request handler</td>
</tr>
<tr>
<td>11.17</td>
<td>CGIHTTPServer</td>
<td>CGI-capable HTTP request handler</td>
</tr>
<tr>
<td>11.18</td>
<td>Cookie</td>
<td>HTTP state management</td>
</tr>
<tr>
<td>11.19</td>
<td>asyncore</td>
<td>Asynchronous socket handler</td>
</tr>
</tbody>
</table>

### 12 Internet Data Handling

<table>
<thead>
<tr>
<th>Section</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1</td>
<td>formatter</td>
<td>Generic output formatting</td>
</tr>
<tr>
<td>12.2</td>
<td>rfc822parser</td>
<td>Parse RFC 822 mail headers</td>
</tr>
<tr>
<td>12.3</td>
<td>mimetypes</td>
<td>Tools for parsing MIME messages</td>
</tr>
<tr>
<td>12.4</td>
<td>MimeWriter</td>
<td>Generic MIME file writer</td>
</tr>
<tr>
<td>12.5</td>
<td>multiline</td>
<td>Support for files containing distinct parts</td>
</tr>
<tr>
<td>12.6</td>
<td>binhex</td>
<td>Encode and decode binhex4 files</td>
</tr>
<tr>
<td>12.7</td>
<td>uu</td>
<td>Encode and decode uuencode files</td>
</tr>
<tr>
<td>12.8</td>
<td>binascii</td>
<td>Convert between binary and ASCII</td>
</tr>
<tr>
<td>12.9</td>
<td>xdrlib</td>
<td>Encode and decode XDR data</td>
</tr>
<tr>
<td>12.10</td>
<td>mailcap</td>
<td>Mailcap file handling</td>
</tr>
<tr>
<td>12.11</td>
<td>mimetypes</td>
<td>Map filenames to MIME types</td>
</tr>
<tr>
<td>12.12</td>
<td>base64</td>
<td>Encode and decode MIME base64 data</td>
</tr>
<tr>
<td>12.13</td>
<td>quopri</td>
<td>Encode and decode MIME quoted-printable data</td>
</tr>
<tr>
<td>12.14</td>
<td>mailbox</td>
<td>Read various mailbox formats</td>
</tr>
<tr>
<td>12.15</td>
<td>mhlib</td>
<td>Access to MH mailboxes</td>
</tr>
<tr>
<td>12.16</td>
<td>mimify</td>
<td>MIME processing of mail messages</td>
</tr>
<tr>
<td>12.17</td>
<td>netrc</td>
<td>netrc file processing</td>
</tr>
<tr>
<td>12.18</td>
<td>robotparser</td>
<td>Parser for robots.txt</td>
</tr>
</tbody>
</table>

### 13 Structured Markup Processing Tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>sgmllib</td>
<td>Simple SGML parser</td>
</tr>
<tr>
<td>13.2</td>
<td>htmllib</td>
<td>A parser for HTML documents</td>
</tr>
<tr>
<td>13.3</td>
<td>htmlentitydefs</td>
<td>Definitions of HTML general entities</td>
</tr>
<tr>
<td>13.4</td>
<td>xml.parsers.expat</td>
<td>Fast XML parsing using Expat</td>
</tr>
<tr>
<td>13.5</td>
<td>xml.dom</td>
<td>The Document Object Model API</td>
</tr>
<tr>
<td>13.6</td>
<td>xml.dom.minidom</td>
<td>Lightweight DOM implementation</td>
</tr>
<tr>
<td>13.7</td>
<td>xml.dom.pulldom</td>
<td>Support for building partial DOM trees</td>
</tr>
<tr>
<td>13.8</td>
<td>xml.sax</td>
<td>Support for SAX2 parsers</td>
</tr>
<tr>
<td>13.9</td>
<td>xml.sax.handler</td>
<td>Base classes for SAX handlers</td>
</tr>
<tr>
<td>13.10</td>
<td>xml.sax.saxutils</td>
<td>SAX Utilities</td>
</tr>
<tr>
<td>13.11</td>
<td>xml.sax.xmlreader</td>
<td>Interface for XML parsers</td>
</tr>
<tr>
<td>13.12</td>
<td>xmllib</td>
<td>A parser for XML documents</td>
</tr>
</tbody>
</table>

### 14 Multimedia Services

<table>
<thead>
<tr>
<th>Section</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
</table>

---
14.1 audioop — Manipulate raw audio data .................................................. 391
14.2 imageop — Manipulate raw image data ............................................... 394
14.3 aifc — Read and write AIFF and AIFC files .................................... 395
14.4 sunau — Read and write Sun AU files ................................................. 397
14.5 wave — Read and write WAV files ....................................................... 399
14.6 chunk — Read IFF chunked data ......................................................... 401
14.7 colorsys — Conversions between color systems ............................... 402
14.8 rgbimg — Read and write “SGI RGB” files ..................................... 403
14.9 imghdr — Determine the type of an image ..................................... 404
14.10 sndhdr — Determine type of sound file ......................................... 404

15 Cryptographic Services ................................................................. 407
  15.1 md5 — MD5 message digest algorithm ............................................. 407
  15.2 sha — SHA message digest algorithm ............................................. 408
  15.3 mpz — GNU arbitrary magnitude integers ....................................... 409
  15.4 rotor — Enigma-like encryption and decryption .............................. 410

16 Restricted Execution ..................................................................... 413
  16.1 rexec — Restricted execution framework ......................................... 414
  16.2 Bastion — Restricting access to objects ......................................... 416

17 Python Language Services ......................................................... 417
  17.1 parser — Access Python parse trees ............................................... 417
  17.2 symbol — Constants used with Python parse trees ......................... 426
  17.3 token — Constants used with Python parse trees ......................... 427
  17.4 keyword — Testing for Python keywords ........................................... 427
  17.5 tokenize — Tokenizer for Python source ......................................... 427
  17.6 tabnanny — Detection of ambiguous indentation ............................ 428
  17.7 pycbr — Python class browser support ........................................... 428
  17.8 py_compile — Compile Python source files .................................... 429
  17.9 compileall — Byte-compile Python libraries .................................. 429
  17.10 dis — Disassembler for Python byte code ..................................... 430

18 SGI IRIX Specific Services ......................................................... 439
  18.1 al — Audio functions on the SGI ..................................................... 439
  18.2 AL — Constants used with the al module ........................................ 441
  18.3 cd — CD-ROM access on SGI systems ............................................ 441
  18.4 fl — FORMS library interface for GUI applications ......................... 445
  18.5 FL — Constants used with the fl module .......................................... 449
  18.6 flp — Functions for loading stored FORMS designs ....................... 450
  18.7 fm — Font Manager interface ........................................................ 450
  18.8 gl — Graphics Library interface ..................................................... 451
  18.9 DEVICE — Constants used with the gl module ................................ 453
  18.10 GL — Constants used with the gl module ........................................ 453
  18.11 imgfile — Support for SGI imglib files ........................................ 453
  18.12 jpeg — Read and write JPEG files ............................................... 454

19 SunOS Specific Services ............................................................ 457
  19.1 sunaudiodev — Access to Sun audio hardware ................................ 457
  19.2 SUNAUDIODEV — Constants used with sunaudiodev ....................... 458

20 MS Windows Specific Services .................................................. 459
  20.1 msvcrt — Useful routines from the MS VC++ runtime ....................... 459
  20.2 _winreg — Windows registry access ............................................... 460
  20.3 winsound — Sound-playing interface for Windows ............................ 465
Introduction

The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an import statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built into the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World-Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out:” it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don’t have to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module random) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter 2, “Built-in Types, Exceptions and Functions,” as the remainder of the manual assumes familiarity with this material.

Let the show begin!
Built-in Types, Exceptions and Functions

Names for built-in exceptions and functions are found in a separate symbol table. This table is searched last when the interpreter looks up the meaning of a name, so local and global user-defined names can override built-in names. Built-in types are described together here for easy reference.\(^1\)

The tables in this chapter document the priorities of operators by listing them in order of ascending priority (within a table) and grouping operators that have the same priority in the same box. Binary operators of the same priority group from left to right. (Unary operators group from right to left, but there you have no real choice.) See chapter 5 of the Python Reference Manual for the complete picture on operator priorities.

2.1 Built-in Types

The following sections describe the standard types that are built into the interpreter. These are the numeric types, sequence types, and several others, including types themselves. There is no explicit Boolean type; use integers instead.

Some operations are supported by several object types; in particular, all objects can be compared, tested for truth value, and converted to a string (with the ‘...’ notation). The latter conversion is implicitly used when an object is written by the print statement.

2.1.1 Truth Value Testing

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below. The following values are considered false:

- None
- zero of any numeric type, for example, 0, 0L, 0.0, 0j.
- any empty sequence, for example, ’’, (), [].
- any empty mapping, for example, {}.
- instances of user-defined classes, if the class defines a __nonzero__() or __len__() method, when that method returns zero.\(^2\)

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return 0 for false and 1 for true, unless otherwise stated. (Important exception: the Boolean operations ‘or’ and ‘and’ always return one of their operands.)

\(^1\)Most descriptions sorely lack explanations of the exceptions that may be raised — this will be fixed in a future version of this manual.

\(^2\)Additional information on these special methods may be found in the Python Reference Manual.
2.1.2 Boolean Operations

These are the Boolean operations, ordered by ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \lor y$</td>
<td>if $x$ is false, then $y$, else $x$</td>
<td>(1)</td>
</tr>
<tr>
<td>$x \land y$</td>
<td>if $x$ is false, then $x$, else $y$</td>
<td>(1)</td>
</tr>
<tr>
<td>$\neg x$</td>
<td>if $x$ is false, then 1, else 0</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

(1) These only evaluate their second argument if needed for their outcome.

(2) ‘$\neg$’ has a lower priority than non-Boolean operators, so $\neg a == b$ is interpreted as $\neg (a == b)$, and $a == \neg b$ is a syntax error.

2.1.3 Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, $x < y \leq z$ is equivalent to $x < y$ and $y \leq z$, except that $y$ is evaluated only once (but in both cases $z$ is not evaluated at all when $x < y$ is found to be false).

This table summarizes the comparison operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;$</td>
<td>strictly less than</td>
<td></td>
</tr>
<tr>
<td>$\leq$</td>
<td>less than or equal</td>
<td></td>
</tr>
<tr>
<td>$&gt;$</td>
<td>strictly greater than</td>
<td></td>
</tr>
<tr>
<td>$\geq$</td>
<td>greater than or equal</td>
<td></td>
</tr>
<tr>
<td>$==$</td>
<td>equal</td>
<td></td>
</tr>
<tr>
<td>$!=$</td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td>$&lt;&gt;$</td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td>$is$</td>
<td>object identity</td>
<td></td>
</tr>
<tr>
<td>$is \ not$</td>
<td>negated object identity</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) $<>$ and $!= $ are alternate spellings for the same operator. (I couldn’t choose between ABC and C! :-) $!= $ is the preferred spelling; $<>$ is obsolescent.

Objects of different types, except different numeric types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (for example, file objects) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently.

Instances of a class normally compare as non-equal unless the class defines the __cmp__ method. Refer to the Python Reference Manual for information on the use of this method to effect object comparisons.

**Implementation note:** Objects of different types except numbers are ordered by their type names; objects of the same types that don’t support proper comparison are ordered by their address.

Two more operations with the same syntactic priority, ‘in’ and ‘not in’, are supported only by sequence types (below).
2.1.4 Numeric Types

There are four numeric types: plain integers, long integers, floating point numbers, and complex numbers. Plain integers (also just called integers) are implemented using long in C, which gives them at least 32 bits of precision. Long integers have unlimited precision. Floating point numbers are implemented using double in C. All bets on their precision are off unless you happen to know the machine you are working with.

Complex numbers have a real and imaginary part, which are both implemented using double in C. To extract these parts from a complex number \( z \), use \( z.\text{real} \) and \( z.\text{imag} \).

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex and octal numbers) yield plain integers. Integer literals with an ‘L’ or ‘l’ suffix yield long integers (‘L’ is preferred because ‘1l’ looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending ‘j’ or ‘J’ to a numeric literal yields a complex number.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “smaller” type is converted to that of the other, where plain integer is smaller than long integer is smaller than floating point is smaller than complex. Comparisons between numbers of mixed type use the same rule.³

The functions \( \text{int()} \), \( \text{long()} \), \( \text{float()} \), and \( \text{complex()} \) can be used to coerce numbers to a specific type.

All numeric types support the following operations, sorted by ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x + y )</td>
<td>sum of ( x ) and ( y )</td>
<td></td>
</tr>
<tr>
<td>( x - y )</td>
<td>difference of ( x ) and ( y )</td>
<td></td>
</tr>
<tr>
<td>( x * y )</td>
<td>product of ( x ) and ( y )</td>
<td></td>
</tr>
<tr>
<td>( x \div y )</td>
<td>quotient of ( x ) and ( y )</td>
<td>(1)</td>
</tr>
<tr>
<td>( x % y )</td>
<td>remainder of ( x / y )</td>
<td></td>
</tr>
<tr>
<td>( -x )</td>
<td>( x ) negated</td>
<td></td>
</tr>
<tr>
<td>(+x)</td>
<td>( x ) unchanged</td>
<td></td>
</tr>
<tr>
<td>abs((x))</td>
<td>absolute value or magnitude of ( x )</td>
<td>(2)</td>
</tr>
<tr>
<td>int((x))</td>
<td>( x ) converted to integer</td>
<td></td>
</tr>
<tr>
<td>long((x))</td>
<td>( x ) converted to long integer</td>
<td>(2)</td>
</tr>
<tr>
<td>float((x))</td>
<td>( x ) converted to floating point</td>
<td></td>
</tr>
<tr>
<td>complex((re, im))</td>
<td>a complex number with real part ( re ), imaginary part ( im ). ( im ) defaults to zero.</td>
<td></td>
</tr>
<tr>
<td>c.conjugate()</td>
<td>conjugate of the complex number ( c )</td>
<td></td>
</tr>
<tr>
<td>divmod((x, y))</td>
<td>the pair ( (x / y, x % y) )</td>
<td>(3)</td>
</tr>
<tr>
<td>pow((x, y))</td>
<td>( x ) to the power ( y )</td>
<td></td>
</tr>
<tr>
<td>( x ** y )</td>
<td>( x ) to the power ( y )</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: \( 1/2 \) is 0, \((-1)/2 \) is -1, \(1/(-2) \) is -1, and \((-1)/(-2) \) is 0. Note that the result is a long integer if either operand is a long integer, regardless of the numeric value.

(2) Conversion from floating point to (long or plain) integer may round or truncate as in C; see functions \( \text{floor()} \) and \( \text{ceil()} \) in the \text{math} module for well-defined conversions.

(3) See section 2.3, “Built-in Functions,” for a full description.

Bit-string Operations on Integer Types

³As a consequence, the list \([1, 2]\) is considered equal to \([1.0, 2.0]\), and similar for tuples.
Plain and long integer types support additional operations that make sense only for bit-strings. Negative numbers are treated as their 2’s complement value (for long integers, this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bit-wise operations are all lower than the numeric operations and higher than the comparisons; the unary operation ‘˜’ has the same priority as the other unary numeric operations (‘+’ and ‘-’).

This table lists the bit-string operations sorted in ascending priority (operations in the same box have the same priority):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>bitwise or of x and y</td>
</tr>
<tr>
<td>x ˆ y</td>
<td>bitwise exclusive or of x and y</td>
<td></td>
</tr>
<tr>
<td>x &amp; y</td>
<td>bitwise and of x and y</td>
<td></td>
</tr>
<tr>
<td>x &lt;&lt; n</td>
<td>x shifted left by n bits</td>
<td>(1), (2)</td>
</tr>
<tr>
<td>x &gt;&gt; n</td>
<td>x shifted right by n bits</td>
<td>(1), (3)</td>
</tr>
<tr>
<td>˜x</td>
<td>the bits of x inverted</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) Negative shift counts are illegal and cause a ValueError to be raised.

(2) A left shift by n bits is equivalent to multiplication by pow(2, n) without overflow check.

(3) A right shift by n bits is equivalent to division by pow(2, n) without overflow check.

2.1.5 Sequence Types

There are six sequence types: strings, Unicode strings, lists, tuples, buffers, and xrange objects.

Strings literals are written in single or double quotes: ‘xyzzy’, "frobozz". See chapter 2 of the Python Reference Manual for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceeding ‘u’ character: u’abc’, u"def". Lists are constructed with square brackets, separating items with commas: [a, b, c]. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, e.g., a, b, c or (). A single item tuple must have a trailing comma, e.g., (d, ). Buffers are not directly supported by Python syntax, but can be created by calling the built-in function buffer(). XRanges objects are similar to buffers in that there is no specific syntax to create them, but they are created using the xrange() function.

Sequence types support the following operations. The ‘in’ and ‘not in’ operations have the same priorities as the comparison operations. The ‘+’ and ‘*’ operations have the same priority as the corresponding numeric operations.

This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, s and t are sequences of the same type; n, i and j are integers:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>1 if an item of s is equal to x, else 0</td>
<td></td>
</tr>
<tr>
<td>x not in s</td>
<td>0 if an item of s is equal to x, else 1</td>
<td></td>
</tr>
<tr>
<td>s + t</td>
<td>the concatenation of s and t</td>
<td></td>
</tr>
<tr>
<td>s * n, n * s</td>
<td>n copies of s concatenated</td>
<td></td>
</tr>
<tr>
<td>s[i]</td>
<td>i’th item of s, origin 0</td>
<td></td>
</tr>
<tr>
<td>s[i:j]</td>
<td>slice of s from i to j</td>
<td>(2), (3)</td>
</tr>
<tr>
<td>len(s)</td>
<td>length of s</td>
<td></td>
</tr>
<tr>
<td>min(s)</td>
<td>smallest item of s</td>
<td></td>
</tr>
<tr>
<td>max(s)</td>
<td>largest item of s</td>
<td></td>
</tr>
</tbody>
</table>

4 They must have since the parser can’t tell the type of the operands.
Notes:

(1) Values of \( n \) less than 0 are treated as 0 (which yields an empty sequence of the same type as \( s \)).

(2) If \( i \) or \( j \) is negative, the index is relative to the end of the string, i.e., \( \text{len}(s) + i \) or \( \text{len}(s) + j \) is substituted. But note that \(-0\) is still 0.

(3) The slice of \( s \) from \( i \) to \( j \) is defined as the sequence of items with index \( k \) such that \( i \leq k < j \). If \( i \) or \( j \) is greater than \( \text{len}(s) \), use \( \text{len}(s) \). If \( i \) is omitted, use 0. If \( j \) is omitted, use \( \text{len}(s) \). If \( i \) is greater than or equal to \( j \), the slice is empty.

String Methods

These are the string methods which both 8-bit strings and Unicode objects support:

- `capitalize()`
  Return a copy of the string with only its first character capitalized.

- `center(width)`
  Return centered in a string of length \( width \). Padding is done using spaces.

- `count(sub[, start[, end]])`
  Return the number of occurrences of substring \( sub \) in string \( S[start:end] \). Optional arguments \( start \) and \( end \) are interpreted as in slice notation.

- `encode([encoding[, errors]])`
  Return an encoded version of the string. Default encoding is the current default string encoding. \( errors \) may be given to set a different error handling scheme. The default for \( errors \) is ‘strict’, meaning that encoding errors raise a `ValueError`. Other possible values are ‘ignore’ and ‘replace’. New in version 2.0.

- `endswith(suffix[, start[, end]])`
  Return true if the string ends with the specified \( suffix \), otherwise return false. With optional \( start \), test beginning at that position. With optional \( end \), stop comparing at that position.

- `expandtabs([tabsize])`
  Return a copy of the string where all tab characters are expanded using spaces. If \( tabsize \) is not given, a tab size of 8 characters is assumed.

- `find(sub[, start[, end]])`
  Return the lowest index in the string where substring \( sub \) is found, such that \( sub \) is contained in the range \( [start, end] \). Optional arguments \( start \) and \( end \) are interpreted as in slice notation. Return \(-1\) if \( sub \) is not found.

- `index(sub[, start[, end]])`
  Like `find()`, but raise `ValueError` when the substring is not found.

- `isalnum()`
  Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

- `isalpha()`
  Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

- `isdigit()`
  Return true if there are only digit characters, false otherwise.

- `islower()`
  Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

- `isspace()`
  Return true if there are only whitespace characters in the string and the string is not empty, false otherwise.
istitle()
Return true if the string is a titlecased string, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

isupper()
Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

join(seq)
Return a string which is the concatenation of the strings in the sequence seq. The separator between elements is the string providing this method.

ljust(width)
Return the string left justified in a string of length width. Padding is done using spaces. The original string is returned if width is less than len(s).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

replace(old, new[, maxsplit])
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument maxsplit is given, only the first maxsplit occurrences are replaced.

rfind(sub [,start [,end ]])
Return the highest index in the string where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex(sub [,start [,end ]])
Like rfind() but raises ValueError when the substring sub is not found.

rjust(width)
Return the string right justified in a string of length width. Padding is done using spaces. The original string is returned if width is less than len(s).

rstrip()
Return a copy of the string with trailing whitespace removed.

split([sep [,maxsplit ]])
Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or None, any whitespace string is a separator.

splitlines([keepends])
Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

startswith(prefix [,start [,end ]])
Return true if string starts with the prefix, otherwise return false. With optional start, test string beginning at that position. With optional end, stop comparing string at that position.

strip()
Return a copy of the string with leading and trailing whitespace removed.

swapcase()
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

title()
Return a titlecased version of, i.e. words start with uppercase characters, all remaining cased characters are lowercase.

translate(table[, deletechars ])

Return a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

upper()
Return a copy of the string converted to uppercase.

String Formatting Operations

String and Unicode objects have one unique built-in operation: the % operator (modulo). Given format %values (where format is a string or Unicode object), % conversion specifications in format are replaced with zero or more elements of values. The effect is similar to the using sprintf() in the C language. If format is a Unicode object, or if any of the objects being converted using the %s conversion are Unicode objects, the result will be a Unicode object as well.

If format requires a single argument, values may be a single non-tuple object. Otherwise, values must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The ‘%’ character, which marks the start of the specifier.
2. Mapping key value (optional), consisting of an identifier in parentheses (for example, (asename)).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an ‘*’ (asterisk), the actual width is read from the next element of the tuple in values, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a ‘.’ (dot) followed by the precision. If specified as ‘*’ (an asterisk), the actual width is read from the next element of the tuple in values, and the value to convert comes after the precision.
7. Conversion type.

If the right argument is a dictionary (or any kind of mapping), then the formats in the string must have a parenthesized key into that dictionary inserted immediately after the ‘%’ character, and each format formats the corresponding entry from the mapping. For example:

```python
>>> count = 2
>>> language = 'Python'
>>> print '%(language)s has %(count)03d quote types.' % vars()
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘#’</td>
<td>The value conversion will use the “alternate form” (where defined below).</td>
</tr>
<tr>
<td>‘0’</td>
<td>The conversion will be zero padded.</td>
</tr>
<tr>
<td>‘-‘</td>
<td>The converted value is left adjusted (overrides ‘-‘).</td>
</tr>
<tr>
<td>‘ ‘</td>
<td>(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.</td>
</tr>
<tr>
<td>‘+’</td>
<td>A sign character (+‘ or ‘-‘) will precede the conversion (overrides a ”space” flag).</td>
</tr>
</tbody>
</table>

A tuple object in this case should be a singleton.
The length modifier may be h, i, and L may be present, but are ignored as they are not necessary for Python.

The conversion types are:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>Signed integer decimal.</td>
</tr>
<tr>
<td>'i'</td>
<td>Signed integer decimal.</td>
</tr>
<tr>
<td>'o'</td>
<td>Unsigned octal.</td>
</tr>
<tr>
<td>'u'</td>
<td>Unsigned decimal.</td>
</tr>
<tr>
<td>'x'</td>
<td>Unsigned hexadecimal (lowercase).</td>
</tr>
<tr>
<td>'X'</td>
<td>Unsigned hexadecimal (uppercase).</td>
</tr>
<tr>
<td>'e'</td>
<td>Floating point exponential format (lowercase).</td>
</tr>
<tr>
<td>'E'</td>
<td>Floating point exponential format (uppercase).</td>
</tr>
<tr>
<td>'f'</td>
<td>Floating point decimal format.</td>
</tr>
<tr>
<td>'F'</td>
<td>Floating point decimal format.</td>
</tr>
<tr>
<td>'g'</td>
<td>Same as 'e' if exponent is greater than -4 or less than precision, 'f' otherwise.</td>
</tr>
<tr>
<td>'G'</td>
<td>Same as 'E' if exponent is greater than -4 or less than precision, 'F' otherwise.</td>
</tr>
<tr>
<td>'c'</td>
<td>Single character (accepts integer or single character string).</td>
</tr>
<tr>
<td>'r'</td>
<td>String (converts any python object using \texttt{repr}()).</td>
</tr>
<tr>
<td>'s'</td>
<td>String (converts any python object using \texttt{str}()).</td>
</tr>
</tbody>
</table>
| '%'        | No argument is converted, results in a '‘%’ character in the result. (The complete specification is ‘%’.)

Since Python strings have an explicit length, %s conversions do not assume that ‘\0’ is the end of the string.

For safety reasons, floating point precisions are clipped to 50; %f conversions for numbers whose absolute value is over 1e25 are replaced by %g conversions. All other errors raise exceptions.

Additional string operations are defined in standard module \texttt{string} and in built-in module \texttt{re}.

\textbf{XRange Type}

The \texttt{xrange} type is an immutable sequence which is commonly used for looping. The advantage of the \texttt{xrange} type is that an \texttt{xrange} object will always take the same amount of memory, no matter the size of the range it represents. There are no consistent performance advantages.

\texttt{xrange} objects behave like tuples, and offer a single method:

\texttt{tolist()}  
Return a list object which represents the same values as the \texttt{xrange} object.

\textbf{Mutable Sequence Types}

List objects support additional operations that allow in-place modification of the object. These operations would be supported by other mutable sequence types (when added to the language) as well. Strings and tuples are immutable sequence types and such objects cannot be modified once created. The following operations are defined on mutable sequence types (where \( x \) is an arbitrary object):

\footnote{These numbers are fairly arbitrary. They are intended to avoid printing endless strings of meaningless digits without hampering correct use and without having to know the exact precision of floating point values on a particular machine.}
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s[i] = x)</td>
<td>item (i) of (s) is replaced by (x)</td>
<td></td>
</tr>
<tr>
<td>(s[i:j] = t)</td>
<td>slice of (s) from (i) to (j) is replaced by (t)</td>
<td></td>
</tr>
<tr>
<td>(\text{del } s[i:j])</td>
<td>same as (s[i:j] = [])</td>
<td></td>
</tr>
<tr>
<td>(s.append(x))</td>
<td>same as (s[\text{len}(s):\text{len}(s)] = [x])</td>
<td>(1)</td>
</tr>
<tr>
<td>(s.extend(x))</td>
<td>same as (s[\text{len}(s):\text{len}(s)] = x)</td>
<td>(2)</td>
</tr>
<tr>
<td>(s.count(x))</td>
<td>return number of (i)'s for which (s[i] == x)</td>
<td>(3)</td>
</tr>
<tr>
<td>(s.index(x))</td>
<td>return smallest (i) such that (s[i] == x)</td>
<td>(3)</td>
</tr>
<tr>
<td>(s.insert(i, x))</td>
<td>same as (s[i:i] = [x]) if (i &gt;= 0)</td>
<td></td>
</tr>
<tr>
<td>(s.pop(i))</td>
<td>same as (x = s[i]; \text{del } s[i]; \text{return } x)</td>
<td>(4)</td>
</tr>
<tr>
<td>(s.remove(x))</td>
<td>same as (\text{del } s[s.index(x)])</td>
<td>(3)</td>
</tr>
<tr>
<td>(s.reverse())</td>
<td>reverses the items of (s) in place</td>
<td>(5)</td>
</tr>
<tr>
<td>(s.sort([\text{cmpfunc }]))</td>
<td>sort the items of (s) in place</td>
<td>(5), (6)</td>
</tr>
</tbody>
</table>

Notes:

1. The C implementation of Python has historically accepted multiple parameters and implicitly joined them into a tuple; this no longer works in Python 2.0. Use of this misfeature has been deprecated since Python 1.4.

2. Raises an exception when \(x\) is not a list object. The \(\text{extend()}\) method is experimental and not supported by mutable sequence types other than lists.

3. Raises \(\text{ValueError}\) when \(x\) is not found in \(s\).

4. The \(\text{pop()}\) method is only supported by the list and array types. The optional argument \(i\) defaults to \(-1\), so that by default the last item is removed and returned.

5. The \(\text{sort()}\) and \(\text{reverse()}\) methods modify the list in place for economy of space when sorting or reversing a large list. They don’t return the sorted or reversed list to remind you of this side effect.

6. The \(\text{sort()}\) method takes an optional argument specifying a comparison function of two arguments (list items) which should return \(-1, 0\) or \(1\) depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. Note that this slows the sorting process down considerably; e.g. to sort a list in reverse order it is much faster to use calls to the methods \(\text{sort()}\) and \(\text{reverse()}\) than to use the built-in function \(\text{sort()}\) with a comparison function that reverses the ordering of the elements.

### 2.1.6 Mapping Types

A mapping object maps values of one type (the key type) to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. A dictionary’s keys are almost arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g. \(1\) and \(1.0\)) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are created by placing a comma-separated list of key: value pairs within braces, for example: \{'jack': 4098, 'sjoerd': 4127\} or \{4098: 'jack', 4127: 'sjoerd'\}.

The following operations are defined on mappings (where \(a\) and \(b\) are mappings, \(k\) is a key, and \(v\) and \(x\) are arbitrary objects):
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>the number of items in <code>a</code></td>
<td></td>
</tr>
<tr>
<td><code>a[k]</code></td>
<td>the item of <code>a</code> with key <code>k</code></td>
<td></td>
</tr>
<tr>
<td><code>a[k] = v</code></td>
<td>set <code>a[k]</code> to <code>v</code></td>
<td></td>
</tr>
<tr>
<td><code>del a[k]</code></td>
<td>remove <code>a[k]</code> from <code>a</code></td>
<td></td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td>remove all items from <code>a</code></td>
<td></td>
</tr>
<tr>
<td><code>a.copy()</code></td>
<td>a (shallow) copy of <code>a</code></td>
<td></td>
</tr>
<tr>
<td><code>a.has_key(k)</code></td>
<td>1 if <code>a</code> has a key <code>k</code>, else 0</td>
<td>(1)</td>
</tr>
<tr>
<td><code>a.items()</code></td>
<td>a copy of <code>a</code>'s list of <code>(key, value)</code> pairs</td>
<td></td>
</tr>
<tr>
<td><code>a.keys()</code></td>
<td>a copy of <code>a</code>'s list of keys</td>
<td></td>
</tr>
<tr>
<td><code>a.update(b)</code></td>
<td>for <code>k</code> in <code>b.keys()</code>: <code>a[k] = b[k]</code></td>
<td>(3)</td>
</tr>
<tr>
<td><code>a.values()</code></td>
<td>a copy of <code>a</code>'s list of values</td>
<td></td>
</tr>
<tr>
<td><code>a.get(k, x)</code></td>
<td><code>a[k]</code> if <code>a</code> has key <code>k</code>, else <code>x</code></td>
<td>(4)</td>
</tr>
<tr>
<td><code>a.setdefault(k, x)</code></td>
<td><code>a[k]</code> if <code>a</code> has key <code>k</code>, else <code>x</code> (also setting it)</td>
<td>(5)</td>
</tr>
<tr>
<td><code>a.popitem()</code></td>
<td>remove and return an arbitrary ⟨key, value⟩ pair</td>
<td>(6)</td>
</tr>
</tbody>
</table>

Notes:

(1) Raises a `KeyError` exception if `k` is not in the map.

(2) Keys and values are listed in random order. If `keys()` and `values()` are called with no intervening modifications to the dictionary, the two lists will directly correspond. This allows the creation of ⟨value, key⟩ pairs using `map()`: `pairs = map(None, a.values(), a.keys())`.

(3) `b` must be of the same type as `a`.

(4) Never raises an exception if `k` is not in the map, instead it returns `x`. `x` is optional; when `x` is not provided and `k` is not in the map, `None` is returned.

(5) `setdefault()` is like `get()`, except that if `k` is missing, `x` is both returned and inserted into the dictionary as the value of `k`.

(6) `popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms.

2.1.7 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

Modules

The only special operation on a module is attribute access: `m.name`, where `m` is a module and `name` accesses a name defined in `m`'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named `foo` to exist, rather it requires an (external) `definition` for a module named `foo` somewhere.)

A special member of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (i.e., you can write `m.__dict__['a'] = 1`, which defines `m.a` to be 1, but you can't write `m.__dict__ = {}`).

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/python2.1/os.pyc'>`.

Classes and Class Instances

See chapters 3 and 7 of the Python Reference Manual for these.
Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

The implementation adds two special read-only attributes: `f.func_code` is a function’s `code object` (see below) and `f.func_globals` is the dictionary used as the function’s global namespace (this is the same as `m.__dict__` where `m` is the module in which the function `f` was defined).

Function objects also support getting and setting arbitrary attributes, which can be used to, e.g. attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes. *Note that the current implementation only supports function attributes on functions written in Python. Function attributes on built-ins may be supported in the future.*

Functions have another special attribute `f.__dict__` (a.k.a. `f.func_dict`) which contains the namespace used to support function attributes. `__dict__` can be accessed directly, set to a dictionary object, or `None`. It can also be deleted (but the following two lines are equivalent):

```python
del func.__dict__
func.__dict__ = None
```

Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: `m.im_self` is the object on which the method operates, and `m.im_func` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)`.

Class instance methods are either *bound* or *unbound*, referring to whether the method was accessed through an instance or a class, respectively. When a method is unbound, its `im_self` attribute will be `None` and if called, an explicit `self` object must be passed as the first argument. In this case, `self` must be an instance of the unbound method’s class (or a subclass of that class), otherwise a `TypeError` is raised.

Like function objects, methods objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (i.e. `meth.im_func`), setting method attributes on either bound or unbound methods is disallowed. Attempting to set a method attribute results in a `TypeError` being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```python
class C:
    def method(self):
        pass

c = C()
c.method.im_func.whoami = 'my name is c'
```

See the *Python Reference Manual* for more information.
Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `func_code` attribute.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec` statement or the built-in `eval()` function.

See the *Python Reference Manual* for more information.

Type Objects

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<type ‘int’>`.

The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name).

It is written as `None`.

The Ellipsis Object

This object is used by extended slice notation (see the *Python Reference Manual*). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name).

It is written as `Ellipsis`.

File Objects

File objects are implemented using C’s `stdio` package and can be created with the built-in function `open()` described in section 2.3, “Built-in Functions.” They are also returned by some other built-in functions and methods, e.g., `os.popen()` and `os.fdopen()` and the `makefile()` method of socket objects.

When a file operation fails for an I/O-related reason, the exception `IOError` is raised. This includes situations where the operation is not defined for some reason, like `seek()` on a tty device or writing a file opened for reading.

Files have the following methods:

- **close()**
  Close the file. A closed file cannot be read or written anymore. Any operation which requires that the file be open will raise a `ValueError` after the file has been closed. Calling `close()` more than once is allowed.

- **flush()**
  Flush the internal buffer, like `stdio`'s `fflush()`. This may be a no-op on some file-like objects.

- **isatty()**
  Return true if the file is connected to a tty(-like) device, else false. **Note:** If a file-like object is not associated with a real file, this method should not be implemented.

- **fileno()**
  Return the integer “file descriptor” that is used by the underlying implementation to request I/O operations from
the operating system. This can be useful for other, lower level interfaces that use file descriptors, e.g. module
fcntl or os.read() and friends. Note: File-like objects which do not have a real file descriptor should not provide this method!

**read**([*size*])

Read at most *size* bytes from the file (less if the read hits EOF before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.) Note that this method may call the underlying C function fread() more than once in an effort to acquire as close to *size* bytes as possible.

**readline**([*size*])

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the *size* argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned when EOF is hit immediately. Note: Unlike stdio’s fgets(), the returned string contains null characters ('\0') if they occurred in the input.

**readlines**( [*sizehint*])

Read until EOF using readline() and return a list containing the lines thus read. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totalling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read. Objects implementing a file-like interface may choose to ignore *sizehint* if it cannot be implemented, or cannot be implemented efficiently.

**xreadlines()**

Equivalent to xreadlines.xreadlines(file).

**seek**([offset [, whence ]])

Set the file’s current position, like stdio’s fseek(). The whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end). There is no return value. Note that if the file is opened for appending (mode ‘a’ or ‘a+’), any seek() operations will be undone at the next write. If the file is only opened for writing in append mode (mode ‘a’), this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode ‘a+’).

**tell()**

Return the file’s current position, like stdio’s ftell().

**truncate**( [*size*])

Truncate the file’s size. If the optional *size* argument present, the file is truncated to (at most) that size. The *size* defaults to the current position. Availability of this function depends on the operating system version (for example, not all UNIX versions support this operation).

**write**(str)

Write a string to the file. There is no return value. Note: Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.

**writelines**(list)

Write a list of strings to the file. There is no return value. (The name is intended to match readlines(); writelines() does not add line separators.)

**xreadlines()**

Equivalent to xreadlines.xreadlines(file). (See the xreadlines module for more information.)

File objects also offer a number of other interesting attributes. These are not required for file-like objects, but should be implemented if they make sense for the particular object.

---

7The advantage of leaving the newline on is that an empty string can be returned to mean EOF without being ambiguous. Another advantage is that (in cases where it might matter, e.g. if you want to make an exact copy of a file while scanning its lines) you can tell whether the last line of a file ended in a newline or not (yes this happens!).

---

2.1. Built-in Types
closed
Boolean indicating the current state of the file object. This is a read-only attribute; the close() method changes the value. It may not be available on all file-like objects.

mode
The I/O mode for the file. If the file was created using the open() built-in function, this will be the value of the mode parameter. This is a read-only attribute and may not be present on all file-like objects.

name
If the file object was created using open(), the name of the file. Otherwise, some string that indicates the source of the file object, of the form ‘<...>’. This is a read-only attribute and may not be present on all file-like objects.

softspace
Boolean that indicates whether a space character needs to be printed before another value when using the print statement. Classes that are trying to simulate a file object should also have a writable softspace attribute, which should be initialized to zero. This will be automatic for most classes implemented in Python (care may be needed for objects that override attribute access); types implemented in C will have to provide a writable softspace attribute. Note: This attribute is not used to control the print statement, but to allow the implementation of print to keep track of its internal state.

Internal Objects
See the Python Reference Manual for this information. It describes stack frame objects, traceback objects, and slice objects.

2.1.8 Special Attributes
The implementation adds a few special read-only attributes to several object types, where they are relevant:

__dict__
A dictionary or other mapping object used to store an object’s (writable) attributes.

__methods__
List of the methods of many built-in object types, e.g., []. __methods__ yields ['append', 'count', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']. This usually does not need to be explicitly provided by the object.

__members__
Similar to __methods__, but lists data attributes. This usually does not need to be explicitly provided by the object.

__class__
The class to which a class instance belongs.

__bases__
The tuple of base classes of a class object.

2.2 Built-in Exceptions
Exceptions can be class objects or string objects. Though most exceptions have been string objects in past versions of Python, in Python 1.5 and newer versions, all standard exceptions have been converted to class objects, and users are encouraged to do the same. The exceptions are defined in the module exceptions. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace.
Two distinct string objects with the same value are considered different exceptions. This is done to force programmers to use exception names rather than their string value when specifying exception handlers. The string value of all built-in exceptions is their name, but this is not a requirement for user-defined exceptions or exceptions defined by library modules.

For class exceptions, in a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the raise statement. For string exceptions, the associated value itself will be stored in the variable named as the second argument of the except clause (if any). For class exceptions, that variable receives the exception instance. If the exception class is derived from the standard root class Exception, the associated value is present as the exception instance’s args attribute, and possibly on other attributes as well.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The following exceptions are only used as base classes for other exceptions.

**exception Exception**

The root class for exceptions. All built-in exceptions are derived from this class. All user-defined exceptions should also be derived from this class, but this is not (yet) enforced. The str() function, when applied to an instance of this class (or most derived classes) returns the string value of the argument or arguments, or an empty string if no arguments were given to the constructor. When used as a sequence, this accesses the arguments given to the constructor (handy for backward compatibility with old code). The arguments are also available on the instance’s args attribute, as a tuple.

**exception StandardError**

The base class for all built-in exceptions except SystemExit. StandardError itself is derived from the root class Exception.

**exception ArithmeticError**

The base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError.

**exception LookupError**

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError. This can be raised directly by sys.setdefaultencoding().

**exception EnvironmentError**

The base class for exceptions that can occur outside the Python system: IOError, OSError. When exceptions of this type are created with a 2-tuple, the first item is available on the instance’s errno attribute (it is assumed to be an error number), and the second item is available on the strerror attribute (it is usually the associated error message). The tuple itself is also available on the args attribute. New in version 1.5.2.

When an EnvironmentError exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the filename attribute. However, for backwards compatibility, the args attribute contains only a 2-tuple of the first two constructor arguments.

The filename attribute is None when this exception is created with other than 3 arguments. The errno and strerror attributes are also None when the instance was created with other than 2 or 3 arguments. In this last case, args contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

**exception AssertionError**

Raised when an assert statement fails.
exception AttributeError
Raised when an attribute reference or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)

exception EOFError
Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data. (N.B.: the read() and readline() methods of file objects return an empty string when they hit EOF.)

exception FloatingPointError
Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the --with-fpectl option, or the WANT_SIGFPE_HANDLER symbol is defined in the 'config.h' file.

exception IOError
Raised when an I/O operation (such as a print statement, the built-in open() function or a method of a file object) fails for an I/O-related reason, e.g., “file not found” or “disk full”.

This class is derived from EnvironmentError. See the discussion above for more information on exception instance attributes.

exception ImportError
Raised when an import statement fails to find the module definition or when a from ... import fails to find a name that is to be imported.

exception IndexError
Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, TypeError is raised.)

exception KeyError
Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception KeyboardInterrupt
Raised when the user hits the interrupt key (normally Control-C or DEL). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function input() or raw_input() is waiting for input also raise this exception.

exception MemoryError
Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s malloc() function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError
Raised when a local or global name is not found. This applies only to unqualified names. The associated value is the name that could not be found.

exception NotImplementedError
This exception is derived from RuntimeError. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method. New in version 1.5.2.

exception OSError
This class is derived from EnvironmentError and is used primarily as the os module’s os.error exception. See EnvironmentError above for a description of the possible associated values. New in version 1.5.2.

exception OverflowError
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise MemoryError than give up). Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren’t checked. For plain integers,
all operations that can overflow are checked except left shift, where typical applications prefer to drop bits than raise an exception.

**exception RuntimeError**
Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong. (This exception is mostly a relic from a previous version of the interpreter; it is not used very much any more.)

**exception SyntaxError**
Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to the built-in function `eval()` or `input()`, or when reading the initial script or standard input (also interactively).

When class exceptions are used, instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details; for string exceptions, the associated value is usually a tuple of the form `(message, (filename, lineno, offset, text))`. For class exceptions, `str()` returns only the message.

**exception SystemError**
Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version string of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception’s associated value) and if possible the source of the program that triggered the error.

**exception SystemExit**
This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C’s `exit()` function); if it is None, the exit status is zero; if it has another type (such as a string), the object’s value is printed and the exit status is one.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to None). Also, this exception derives directly from `Exception` and not `StandardError`, since it is not technically an error.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (e.g., after a `fork()` in the child process).

**exception TypeError**
Raised when a built-in operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

**exception UnboundLocalError**
Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`. New in version 2.0.

**exception UnicodeError**
Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`. New in version 2.0.

**exception ValueError**
Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

**exception WindowsError**
Raised when a Windows-specific error occurs or when the error number does not correspond to an `errno` value. The `errno` and `strerror` values are created from the return values of the `GetLastError()` and
FormatException() functions from the Windows Platform API. This is a subclass of OSError. New in version 2.0.

**exception ZeroDivisionError**
Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are used as warning categories; see the warnings module for more information.

**exception Warning**
Base class for warning categories.

**exception UserWarning**
Base class for warnings generated by user code.

**exception DeprecationWarning**
Base class for warnings about deprecated features.

**exception SyntaxWarning**
Base class for warnings about dubious syntax

**exception RuntimeWarning**
Base class for warnings about dubious runtime behavior.

### 2.3 Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

__import__(name[, globals[, locals[, fromlist ]]])
This function is invoked by the import statement. It mainly exists so that you can replace it with another function that has a compatible interface, in order to change the semantics of the import statement. For examples of why and how you would do this, see the standard library modules `ihooks` and `reexec`. See also the built-in module `imp`, which defines some useful operations out of which you can build your own `__import__()` function.

For example, the statement ‘import spam’ results in the following call: `__import__('spam', globals(), locals(), [])`; the statement ‘from spam.ham import eggs’ results in `__import__('spam.ham', globals(), locals(), ['eggs'])`. Note that even though `locals()` and `[‘eggs’]` are passed in as arguments, the `__import__()` function does not set the local variable named `eggs`; this is done by subsequent code that is generated for the import statement. (In fact, the standard implementation does not use its `locals` argument at all, and uses its `globals` only to determine the package context of the import statement.)

When the `name` variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, not the module named by `name`. However, when a non-empty `fromlist` argument is given, the module named by `name` is returned. This is done for compatibility with the bytecode generated for the different kinds of import statement; when using ‘import spam.ham.eggs’, the top-level package `spam` must be placed in the importing namespace, but when using ‘from spam.ham import eggs’, the `spam.ham` subpackage must be used to find the `eggs` variable. As a workaround for this behavior, use `getattr()` to extract the desired components. For example, you could define the following helper:
import string

def my_import(name):
    mod = __import__(name)
    components = string.split(name, '.
    for comp in components[1::]:
        mod = getattr(mod, comp)
    return mod

abs(x)
Return the absolute value of a number. The argument may be a plain or long integer or a floating point number.
If the argument is a complex number, its magnitude is returned.

apply(function, args[, keywords ])
The function argument must be a callable object (a user-defined or built-in function or method, or a class object)
and the args argument must be a sequence (if it is not a tuple, the sequence is first converted to a tuple). The
function is called with args as the argument list; the number of arguments is the the length of the tuple. (This is
different from just calling func(args), since in that case there is always exactly one argument.) If the optional
keywords argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to
be added to the end of the the argument list.

buffer(object[, offset[, size ]])
The object argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers).
A new buffer object will be created which references the object argument. The buffer object will be a slice from
the beginning of object (or from the specified offset). The slice will extend to the end of object (or will have a
length given by the size argument).

callable(object)
Return true if the object argument appears callable, false if not. If this returns true, it is still possible that a call
fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a
new instance); class instances are callable if they have a __call__() method.

chr(i)
Return a string of one character whose ASCII code is the integer i, e.g., chr(97) returns the string ‘a’. This
is the inverse of ord(). The argument must be in the range [0..255], inclusive; ValueError will be raised if
i is outside that range.

cmp(x, y)
Compare the two objects x and y and return an integer according to the outcome. The return value is negative if
x < y, zero if x == y and strictly positive if x > y.

coerce(x, y)
Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as
used by arithmetic operations.

compile(string, filename, kind)
Compile the string into a code object. Code objects can be executed by an exec statement or evaluated by a call
to eval(). The filename argument should give the file from which the code was read; pass e.g. ’<string>’
if it wasn’t read from a file. The kind argument specifies what kind of code must be compiled; it can be ’exec’
if string consists of a sequence of statements, ’eval’ if it consists of a single expression, or ’single’ if
it consists of a single interactive statement (in the latter case, expression statements that evaluate to something
else than None will printed).

complex(real[, imag ])
Create a complex number with the value real + imag*j or convert a string or number to a complex number. Each
argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the function
serves as a numeric conversion function like int(), long() and float(); in this case it also accepts a
string argument which should be a valid complex number.

2.3. Built-in Functions 21
delattr(object, name)
This is a relative of setattr(). The arguments are an object and a string. The string must be the name of one of the object’s attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, 'foobar') is equivalent to del x.foobar.

dir(object)
Without arguments, return the list of names in the current local symbol table. With an argument, attempts to return a list of valid attribute for that object. This information is gleaned from the object’s __dict__, __methods__ and __members__ attributes, if defined. The list is not necessarily complete; e.g., for classes, attributes defined in base classes are not included, and for class instances, methods are not included. The resulting list is sorted alphabetically. For example:

```python
>>> import sys
>>> dir()
['sys']
>>> dir(sys)
['argv', 'exit', 'modules', 'path', 'stderr', 'stdin', 'stdout']
```

divmod(a, b)
Take two numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as (a / b, a % b). For floating point numbers the result is (q, a % b), where q is usually math.floor(a / b) but may be 1 less than that. In any case q * b + a % b is very close to a, if a % b is non-zero it has the same sign as b, and 0 <= abs(a % b) < abs(b).

eval(expression, globals, locals)
The arguments are a string and two optional dictionaries. The expression argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the globals and locals dictionaries as global and local name space. If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where eval is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (e.g. created by compile()). In this case pass a code object instead of a string. The code object must have been compiled passing 'eval' to the kind argument.

Hints: dynamic execution of statements is supported by the exec statement. Execution of statements from a file is supported by the execfile() function. The globals() and locals() functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by eval() or execfile().

execfile(file, globals, locals)
This function is similar to the exec statement, but parses a file instead of a string. It is different from the import statement in that it does not use the module administration — it reads the file unconditionally and does not create a new module.\(^8\)

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the globals and locals dictionaries as global and local namespace. If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed in the environment where execfile() is called. The return value is None.

filter(function, list)

---

\(^8\)It is used relatively rarely so does not warrant being made into a statement.
Construct a list from those elements of `list` for which `function` returns true. If `list` is a string or a tuple, the result also has that type; otherwise it is always a list. If `function` is `None`, the identity function is assumed, i.e. all elements of `list` that are false (zero or empty) are removed.

`float(x)`
Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace; this behaves identical to `string.atof(x)`. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python’s floating point precision) is returned.

Note: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

`getattr(object, name[, default])`
Return the value of the named attributed of `object`. `name` must be a string. If the string is the name of one of the object’s attributes, the result is the value of that attribute. For example, `getattr(x, ‘foobar’)` is equivalent to `x.foobar`. If the named attribute does not exist, `default` is returned if provided, otherwise `AttributeError` is raised.

`globals()`
Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

`hasattr(object, name)`
The arguments are an object and a string. The result is 1 if the string is the name of one of the object’s attributes, 0 if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

`hash(object)`
Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, e.g. 1 and 1.0).

`hex(x)`
Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression. Note: this always yields an unsigned literal, e.g. on a 32-bit machine, `hex(-1)` yields `'0xffffffff'`. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an `OverflowError` exception.

`id(object)`
Return the ‘identity’ of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects whose lifetimes are disjunct may have the same `id()` value. (Implementation note: this is the address of the object.)

`input([prompt])`
Equivalent to `eval(raw_input(prompt))`. **Warning:** This function is not safe from user errors! It expects a valid Python expression as input; if the input is not syntactically valid, a `SyntaxError` will be raised. Other exceptions may be raised if there is an error during evaluation. (On the other hand, sometimes this is exactly what you need when writing a quick script for expert use.)

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Consider using the `raw_input()` function for general input from users.

`int(x[, radix])`
Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace; this behaves identical to `string.atoi(x[, radix])`. The `radix` parameter gives the base for the conversion and may be any integer.
in the range \([2, 36]\), or zero. If \(\text{radix}\) is zero, the proper radix is guessed based on the contents of string; the interpretation is the same as for integer literals. If \(\text{radix}\) is specified and \(x\) is not a string, \(\text{TypeError}\) is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers is defined by the C semantics; normally the conversion truncates towards zero.\(^9\)

\[\text{intern}(\text{string})\]
Enter \text{string} in the table of “interned” strings and return the interned string – which is \text{string} itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys. Interned strings are immortal (i.e. never get garbage collected).

\[\text{isinstance}(\text{object}, \text{class})\]
Return true if the \text{object} argument is an instance of the \text{class} argument, or of a (direct or indirect) subclass thereof. Also return true if \text{class} is a type object and \text{object} is an object of that type. If \text{object} is not a class instance or a object of the given type, the function always returns false. If \text{class} is neither a class object nor a type object, a \(\text{TypeError}\) exception is raised.

\[\text{issubclass}(\text{class1}, \text{class2})\]
Return true if \text{class1} is a subclass (direct or indirect) of \text{class2}. A class is considered a subclass of itself. If either argument is not a class object or a object of the given type, the function always returns false. If \text{class} is neither a class object nor a type object, a \(\text{TypeError}\) exception is raised.

\[\text{len}(s)\]
Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

\[\text{list}(\text{sequence})\]
Return a list whose items are the same and in the same order as \text{sequence}’s items. If \text{sequence} is already a list, a copy is made and returned, similar to \text{sequence}[:]. For instance, \text{list()} \{’abc’\} returns returns \{’a’, ’b’, ’c’\} and \text{list()} \{(1, 2, 3)\} returns \{1, 2, 3\}.

\[\text{locals}()\]
Return a dictionary representing the current local symbol table. Warning: The contents of this dictionary should not be modified; changes may not affect the values of local variables used by the interpreter.

\[\text{long}(x[, \text{radix}])\]
Convert a string or number to a long integer. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace; this behaves identical to \text{string.atol}(x)\. The \text{radix} argument is interpreted in the same way as for \text{int()}, and may only be given when \(x\) is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers is defined by the C semantics; see the description of \text{int()}.

\[\text{map}(\text{function}, \text{list}, ...)\]
Apply \text{function} to every item of \text{list} and return a list of the results. If additional \text{list} arguments are passed, \text{function} must take that many arguments and is applied to the items of all lists in parallel; if a list is shorter than another it is assumed to be extended with \text{None} items. If \text{function} is \text{None}, the identity function is assumed; if there are multiple list arguments, \text{map()} returns a list consisting of tuples containing the corresponding items from all lists (i.e. a kind of transpose operation). The \text{list} arguments may be any kind of sequence; the result is always a list.

\[\text{max}(s[, \text{args}...])\]
With a single argument \(s\), return the largest item of a non-empty sequence (e.g., a string, tuple or list). With more than one argument, return the largest of the arguments.

\[\text{min}(s[, \text{args}...])\]
With a single argument \(s\), return the smallest item of a non-empty sequence (e.g., a string, tuple or list). With

\(^9\)This is ugly — the language definition should require truncation towards zero.
more than one argument, return the smallest of the arguments.

\texttt{oct(x)}

Convert an integer number (of any size) to an octal string. The result is a valid Python expression. Note: this always yields an unsigned literal, e.g. on a 32-bit machine, \texttt{oct(-1)} yields ‘037777777777’. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an OverflowError exception.

\texttt{open(filename[, mode[, bufsize]])}

Return a new file object (described earlier under Built-in Types). The first two arguments are the same as for \texttt{stdio’s fopen()}: \texttt{filename} is the file name to be opened, \texttt{mode} indicates how the file is to be opened: ‘r’ for reading, ‘w’ for writing (truncating an existing file), and ‘a’ opens it for appending (which on some UNIX systems means that all writes append to the end of the file, regardless of the current seek position).

Modes ‘r+’, ‘w+’ and ‘a+’ open the file for updating (note that ‘w+’ truncates the file). Append ‘b’ to the mode to open the file in binary mode, on systems that differentiate between binary and text files (else it is ignored). If the file cannot be opened, IOError is raised.

If \texttt{mode} is omitted, it defaults to ‘r’. When opening a binary file, you should append ‘b’ to the \texttt{mode} value for improved portability. (It’s useful even on systems which don’t treat binary and text files differently, where it serves as documentation.) The optional \texttt{bufsize} argument specifies the file’s desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative \texttt{bufsize} means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used.\footnote{Specifying a buffer size currently has no effect on systems that don’t have \texttt{setvbuf()}. The interface to specify the buffer size is not done using a method that calls \texttt{setvbuf()}, because that may dump core when called after any I/O has been performed, and there’s no reliable way to determine whether this is the case.}

\texttt{ord(c)}

Return the ASCII value of a string of one character or a Unicode character. E.g., \texttt{ord(‘a’)} returns the integer 97, \texttt{ord(u’u2020’)} returns 8224. This is the inverse of \texttt{chr()} for strings and of \texttt{unichr()} for Unicode characters.

\texttt{pow(x, y[, z])}

Return \(x\) to the power \(y\); if \(z\) is present, return \(x\) to the power \(y\), modulo \(z\) (computed more efficiently than \texttt{pow(x, y) \% z}). The arguments must have numeric types. With mixed operand types, the rules for binary arithmetic operators apply. The effective operand type is also the type of the result; if the result is not expressible in this type, the function raises an exception; e.g., \texttt{pow(2, -1)} or \texttt{pow(2, 35000)} is not allowed.

\texttt{range([start,] stop[, step])}

This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the \texttt{step} argument is omitted, it defaults to 1. If the \texttt{start} argument is omitted, it defaults to 0. The full form returns a list of plain integers \([\texttt{start, start + step, start + 2 * step, \ldots}].\) If \texttt{step} is positive, the last element is the largest \texttt{start + i * step} less than \texttt{stop}; if \texttt{step} is negative, the last element is the largest \texttt{start + i * step} greater than \texttt{stop}. \texttt{step} must not be zero (or else ValueError is raised). Example:
```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

**raw_input ([prompt])**

If the `prompt` argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```python
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `raw_input()` will use it to provide elaborate line editing and history features.

**reduce (function, sequence[, initializer])**

Apply `function` of two arguments cumulatively to the items of `sequence`, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `(((1+2)+3)+4)+5)`. If the optional `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

**reload (module)**

Re-parse and re-initialize an already imported `module`. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (i.e. the same as the `module` argument).

There are a number of caveats:

If a module is syntactically correct but its initialization fails, the first `import` statement for it does not bind its name locally, but does store a (partially initialized) module object in `sys.modules`. To reload the module you must first `import` it again (this will bind the name to the partially initialized module object) before you can `reload()` it.

When a module is reloaded, its dictionary (containing the module’s global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module’s advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table’s presence and skip its initialization if desired.

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__` and `__builtin__`. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...,` calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from...import...` statements.
statement, another is to use `import` and qualified names (`module.name`) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

- **`repr(object)`**
  Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`.

- **`round(x[, n])`**
  Return the floating point value `x` rounded to `n` digits after the decimal point. If `n` is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus `n`; if two multiples are equally close, rounding is done away from 0 (so e.g. `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

- **`setattr(object, name, value)`**
  This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

- **`slice([start,] stop[, step])`**
  Return a slice object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to None. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used, e.g. for `a[start:stop:step]` or `a[start:stop, i]`.

- **`str(object)`**
  Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to `eval()`: its goal is to return a printable string.

- **`tuple(sequence)`**
  Return a tuple whose items are the same and in the same order as `sequence`'s items. If `sequence` is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns returns (`'a'`, `'b'`, `'c'`) and `tuple([1, 2, 3])` returns (1, 2, 3).

- **`type(object)`**
  Return the type of an `object`. The return value is a type object. The standard module `types` defines names for all built-in types. For instance:

  ```
  >>> import types
  >>> if type(x) == types.StringType: print "It’s a string"
  ```

- **`unichr(i)`**
  Return the Unicode string of one character whose Unicode code is the integer `i`, e.g., `unichr(97)` returns the string `u’a’`. This is the inverse of `ord()` for Unicode strings. The argument must be in the range [0..65535], inclusive. `ValueError` is raised otherwise. New in version 2.0.

- **`unicode(string[, encoding[, errors]])`**
  Decodes `string` using the codec for `encoding`. Error handling is done according to `errors`. The default behavior is to decode UTF-8 in strict mode, meaning that encoding errors raise `ValueError`. See also the `codecs` module. New in version 2.0.

- **`vars([object])`**
  Without arguments, return a dictionary corresponding to the current local symbol table. With a module, class
or class instance object as argument (or anything else that has a \_dict\_ attribute), returns a dictionary corresponding to the object’s symbol table. The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined.\(^{11}\)

\textbf{xrange}(\texttt{[start, ]stop[, step]})

This function is very similar to \texttt{range()}, but returns an “\texttt{xrange} object” instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of \texttt{xrange()} over \texttt{range()} is minimal (since \texttt{xrange()} still has to create the values when asked for them) except when a very large range is used on a memory-starved machine (e.g. MS-DOS) or when all of the range’s elements are never used (e.g. when the loop is usually terminated with \texttt{break}).

\textbf{zip}(\texttt{seq1}, ...)

This function returns a list of tuples, where each tuple contains the \texttt{i}-th element from each of the argument sequences. At least one sequence is required, otherwise a \texttt{TypeError} is raised. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple argument sequences which are all of the same length, \texttt{zip()} is similar to \texttt{map()} with an initial argument of \texttt{None}. With a single sequence argument, it returns a list of 1-tuples. New in version 2.0.

\(^{11}\)In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (e.g. modules) can be. This may change.
CHAPTER
THREE

Python Runtime Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here’s an overview:

- sys: Access system-specific parameters and functions.
- gc: Interface to the cycle-detecting garbage collector.
- weakref: Support for weak references and weak dictionaries.
- fpectl: Provide control for floating point exception handling.
- atexit: Register and execute cleanup functions.
- types: Names for all built-in types.
- UserDict: Class wrapper for dictionary objects.
- UserList: Class wrapper for list objects.
- UserString: Class wrapper for string objects.
- operator: All Python’s standard operators as built-in functions.
- inspect: Extract information and source code from live objects.
- traceback: Print or retrieve a stack traceback.
- linecache: This module provides random access to individual lines from text files.
- pickle: Convert Python objects to streams of bytes and back.
- cPickle: Faster version of pickle, but not subclassable.
- copy_reg: Register pickle support functions.
- shelf: Python object persistence.
- copy: Shallow and deep copy operations.
- marshal: Convert Python objects to streams of bytes and back (with different constraints).
- warnings: Issue warning messages and control their disposition.
- imp: Access the implementation of the import statement.
- code: Base classes for interactive Python interpreters.
- codeop: Compile (possibly incomplete) Python code.
- pprint: Data pretty printer.
- repr: Alternate repr() implementation with size limits.
- new: Interface to the creation of runtime implementation objects.
- site: A standard way to reference site-specific modules.
- user: A standard way to reference user-specific modules.
- __builtin__: The set of built-in functions.
- __main__: The environment where the top-level script is run.

3.1 sys — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

argv
The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `-c`. If no script name was passed to the Python interpreter, `argv` has zero length.

**byteorder**

An indicator of the native byte order. This will have the value ‘big’ on big-endian (most-significant byte first) platforms, and ‘little’ on little-endian (least-significant byte first) platforms. New in version 2.0.

**builtin_module_names**

A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `modules.keys()` only lists the imported modules.)

**copyright**

A string containing the copyright pertaining to the Python interpreter.

**dllhandle**

Integer specifying the handle of the Python DLL. Availability: Windows.

**displayhook**(value)

If `value` is not `None`, this function prints it to `sys.stdout`, and saves it in `__builtin__.displayhook`.

`sys.displayhook` is called on the result of evaluating an expression entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

**excepthook**(type, value, traceback)

This function prints out a given traceback and exception to `sys.stderr`.

When an exception is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

**__displayhook__**

**__excepthook__**

These objects contain the original values of `displayhook` and `excepthook` at the start of the program. They are saved so that `displayhook` and `excepthook` can be restored in case they happen to get replaced with broken objects.

**exc_info()**

This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, “handling an exception” is defined as “executing or having executed an except clause.” For any stack frame, only information about the most recently handled exception is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values returned are `(type, value, traceback)`. Their meaning is: `type` gets the exception type of the exception being handled (a string or class object); `value` gets the exception parameter (its associated value or the second argument to `raise`, which is always a class instance if the exception type is a class object); `traceback` gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

**Warning:** assigning the `traceback` return value to a local variable in a function that is handling an exception will cause a circular reference. This will prevent anything referenced by a local variable in the same function or by the traceback from being garbage collected. Since most functions don’t need access to the traceback, the best solution is to use something like `type, value = sys.exc_info()[2]` to extract only the exception type and value. If you do need the traceback, make sure to delete it after use (best done with `try ... finally`
statement) or to call exc_info() in a function that does not itself handle an exception.

```python
exc_type
exc_value
exc_traceback
```

**Deprecated since release 1.5.** Use exc_info() instead.

Since they are global variables, they are not specific to the current thread, so their use is not safe in a multi-threaded program. When no exception is being handled, exc_type is set to None and the other two are undefined.

```python
exec_prefix
```

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the --exec-prefix argument to the configure script. Specifically, all configuration files (e.g. the 'config.h' header file) are installed in the directory exec_prefix + '/lib/pythonversion/config', and shared library modules are installed in exec_prefix + '/lib/pythonversion/lib-dynload', where version is equal to version[:3].

```python
executable
```

A string giving the name of the executable binary for the Python interpreter, on systems where this makes sense.

```python
exit([arg])
```

Exit from Python. This is implemented by raising the SystemExit exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level. The optional argument arg can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered “successful termination” and any nonzero value is considered “abnormal termination” by shells and the like. Most systems require it to be in the range 0-127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped: Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed to sys.stderr and results in an exit code of 1. In particular, sys.exit("some error message") is a quick way to exit a program when an error occurs.

```python
exitfunc
```

This value is not actually defined by the module, but can be set by the user (or by a program) to specify a cleanup action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits. Only one function may be installed in this way; to allow multiple functions which will be called at termination, use the atexit module. Note: the exit function is not called when the program is killed by a signal, when a Python fatal internal error is detected, or when os._exit() is called.

```python
getdefaultencoding()
```

Return the name of the current default string encoding used by the Unicode implementation. New in version 2.0.

```python
getrefcount(object)
```

Return the reference count of the object. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to getrefcount().

```python
getrecursionlimit()
```

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by setrecursionlimit().

```python
_getframe([depth])
```

Return a frame object from the call stack. If optional integer depth is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, ValueError is raised. The default for depth is zero, returning the frame at the top of the call stack.

This function should be used for internal and specialized purposes only.

```python
hexversion
```

3.1. sys — System-specific parameters and functions 31
The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```python
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
...
else:
    # use an alternative implementation or warn the user
...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in `hex()` function. The `version_info` value may be used for a more human-friendly encoding of the same information. New in version 1.5.2.

**last_type**

**last_value**

**last_traceback**

These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see the chapter “The Python Debugger” for more information.)

The meaning of the variables is the same as that of the return values from `exc_info()` above. (Since there is only one interactive thread, thread-safety is not a concern for these variables, unlike for `exc_type` etc.)

**maxint**

The largest positive integer supported by Python’s regular integer type. This is at least 2**31-1. The largest negative integer is `-maxint-1` – the asymmetry results from the use of 2’s complement binary arithmetic.

**modules**

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. Note that removing a module from this dictionary is not the same as calling `reload()` on the corresponding module object.

**path**

A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, or an installation-dependent default.

The first item of this list, `path[0]`, is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), `path[0]` is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of PYTHONPATH.

**platform**

This string contains a platform identifier, e.g. ‘sunos5’ or ‘linux1’. This can be used to append platform-specific components to `path`, for instance.

**prefix**

A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string ‘/usr/local’. This can be set at build time with the `--prefix` argument to the `configure` script. The main collection of Python library modules is installed in the directory `prefix + '/lib/pythonversion'` while the platform independent header files (all except ‘config.h’) are stored in `prefix + '/include/pythonversion'`, where `version` is equal to `version[:3]`.

**ps1**

**ps2**

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter
is in interactive mode. Their initial values in this case are `>>>` and `...`. If a non-string object is assigned to either variable, its `str()` is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

**setcheckinterval** *(interval)*
Set the interpreter’s “check interval”. This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 10, meaning the check is performed every 10 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

**setdefaultencoding** *(name)*
Set the current default string encoding used by the Unicode implementation. If name does not match any available encoding, `LookupError` is raised. This function is only intended to be used by the `site` module implementation and, where needed, by `sitecustomize`. Once used by the `site` module, it is removed from the `sys` module’s namespace. New in version 2.0.

**setprofile** *(profilefunc)*
Set the system’s profile function, which allows you to implement a Python source code profiler in Python. See the chapter on the Python Profiler. The system’s profile function is called similarly to the system’s trace function (see `settrace()`), but it isn’t called for each executed line of code (only on call and return and when an exception occurs). Also, its return value is not used, so it can just return `None`.

**setrecursionlimit** *(limit)*
Set the maximum depth of the Python interpreter stack to limit. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when she has a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

**settrace** *(tracefunc)*
Set the system’s trace function, which allows you to implement a Python source code debugger in Python. See section “How It Works” in the chapter on the Python Debugger.

**stdin**
**stdout**
**stderr**

File objects corresponding to the interpreter’s standard input, output and error streams. `stdin` is used for all interpreter input except for scripts but including calls to `input()` and `raw_input()`. `stdout` is used for the output of `print` and expression statements and for the prompts of `input()` and `raw_input()`. The interpreter’s own prompts and (almost all of) its error messages go to `stderr`. `stdout` and `stderr` needn’t be built-in file objects: any object is acceptable as long as it has a `write()` method that takes a string argument. (Changing these objects doesn’t affect the standard I/O streams of processes executed by `os.popen()`, `os.system()` or the `exec*()` family of functions in the `os` module.)

**__stdin__**
**__stdout__**
**__stderr__**

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to restore the actual files to known working file objects in case they have been overwritten with a broken object.

**tracebacklimit**
When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is 1000. When set to 0 or less, all traceback information is suppressed and only the exception type and value are printed.

**version**
A string containing the version number of the Python interpreter plus additional information on the build num-
ber and compiler used. It has a value of the form ‘version (\#build_number, build_date, build_time) [compiler]’. The first three characters are used to identify the version in the installation directories (where appropriate on each platform). An example:

```python
>>> import sys
>>> sys.version
'1.5.2 (#0 Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]'
```

**version_info**

A tuple containing the five components of the version number: major, minor, micro, releaselevel, and serial. All values except releaselevel are integers; the release level is ‘alpha’, ‘beta’, ‘candidate’, or ‘final’. The version_info value corresponding to the Python version 2.0 is (2, 0, 0, ‘final’, 0). New in version 2.0.

**winver**

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of version. It is provided in the sys module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

### 3.2 gc — Garbage Collector interface

The gc module is only available if the interpreter was built with the optional cyclic garbage detector (enabled by default). If this was not enabled, an ImportError is raised by attempts to import this module.

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling gc.disable(). To debug a leaking program call gc.set_debug(gc.DEBUG_LEAK).

The gc module provides the following functions:

**enable()**

Enable automatic garbage collection.

**disable()**

Disable automatic garbage collection.

**isenabled()**

Returns true if automatic collection is enabled.

**collect()**

Run a full collection. All generations are examined and the number of unreachable objects found is returned.

**set_debug(flags)**

Set the garbage collection debugging flags. Debugging information will be written to sys.stderr. See below for a list of debugging flags which can be combined using bit operations to control debugging.

**get_debug()**

Return the debugging flags currently set.

**set_threshold(threshold0, threshold1, threshold2)**

Set the garbage collection thresholds (the collection frequency). Setting threshold0 to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations.
and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds \( \text{threshold0} \), collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than \( \text{threshold1} \) times since generation 1 has been examined, then generation 1 is examined as well. Similarly, \( \text{threshold2} \) controls the number of collections of generation 1 before collecting generation 2.

\[ \text{get_threshold()} \]

Return the current collection thresholds as a tuple of \( (\text{threshold0}, \text{threshold1}, \text{threshold2}) \).

The following variable is provided for read-only access:

\[ \text{garbage} \]

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Objects that have \( \_\_\text{del}\_\_() \) methods and create part of a reference cycle cause the entire reference cycle to be uncollectable. If \text{DEBUG}\_\text{SAVEALL} is set, then all unreachable objects will be added to this list rather than freed.

The following constants are provided for use with \text{set}\_\text{debug}():

\[ \text{DEBUG STATS} \]

Print statistics during collection. This information can be useful when tuning the collection frequency.

\[ \text{DEBUG COLLECTABLE} \]

Print information on collectable objects found.

\[ \text{DEBUG UNCOLLECTABLE} \]

Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the \text{garbage} list.

\[ \text{DEBUG INSTANCES} \]

When \text{DEBUG}\_\text{COLLECTABLE} or \text{DEBUG}\_\text{UNCOLLECTABLE} is set, print information about instance objects found.

\[ \text{DEBUG OBJECTS} \]

When \text{DEBUG}\_\text{COLLECTABLE} or \text{DEBUG}\_\text{UNCOLLECTABLE} is set, print information about objects other than instance objects found.

\[ \text{DEBUG SAVEALL} \]

When set, all unreachable objects found will be appended to \text{garbage} rather than being freed. This can be useful for debugging a leaking program.

\[ \text{DEBUG LEAK} \]

The debugging flags necessary for the collector to print information about a leaking program (equal to \text{DEBUG}\_\text{COLLECTABLE} | \text{DEBUG}\_\text{UNCOLLECTABLE} | \text{DEBUG}\_\text{INSTANCES} | \text{DEBUG}\_\text{OBJECTS} | \text{DEBUG}\_\text{SAVEALL}).

### 3.3 weakref — Weak references

New in version 2.1.

The \text{weakref} module allows the Python programmer to create weak references to objects.

XXX — need to say more here!

Not all objects can be weakly referenced; those objects which do include class instances, functions written in Python (but not in C), and methods (both bound and unbound). Extension types can easily be made to support weak references; see section 3.3.3, “Weak References in Extension Types,” for more information.

\[ \text{ref}(\text{object}[, \text{callback}]) \]

Return a weak reference to \text{object}. If \text{callback} is provided, it will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no
longer be available. The original object can be retrieved by calling the reference object, if the referent is still
alive.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each
weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are
handled in exactly the same way as exceptions raised from an object’s __del__() method.

Weak references are hashable if the object is hashable. They will maintain their hash value even after the object
was deleted. If hash() is called the first time only after the object was deleted, the call will raise TypeError.
Weak references support test for equality, but not ordering. If the object is still alive, to references are equal
if the objects are equal (regardless of the callback). If the object has been deleted, they are equal iff they are
identical.

proxy(object[, callback])
Return a proxy to object which uses a weak reference. This supports use of the proxy in most contexts instead
of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type
of either ProxyType or CallableProxyType, depending on whether object is callable. Proxy objects are
not hashable regardless of the referent; this avoids a number of problems related to their fundamentally mutable
nature, and prevent their use as dictionary keys. callable is the same as the parameter of the same name to the
ref() function.

getweakrefcount(object)
Return the number of weak references and proxies which refer to object.

getweakrefs(object)
Return a list of all weak reference and proxy objects which refer to object.

class WeakKeyDictionary(dict)
Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a
strong reference to the key. This can be used to associate additional data with an object owned by other parts of
an application without adding attributes to those objects. This can be especially useful with objects that override
attribute accesses.

class WeakValueDictionary(dict)
Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong refer-
ence to the value exists anymore.

ReferenceType
The type object for weak references objects.

ProxyType
The type object for proxies of objects which are not callable.

CallableProxyType
The type object for proxies of callable objects.

ProxyTypes
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy
without being dependent on naming both proxy types.

exception ReferenceError
Exception raised when a proxy object is used but the underlying object has been collected.

See Also:

PEP 0205, “Weak References”
The proposal and rationale for this feature, including links to earlier implementations and information about
similar features in other languages.
3.3.1 Weak Reference Objects

Weak reference objects have no attributes or methods, but do allow the referent to be obtained, if it still exists, by calling it:

```python
>>> import weakref
>>> class Object:
...     pass
... >>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
1
```

If the referent no longer exists, calling the reference object returns `None`:

```python
>>> del o, o2
>>> print r()
None
```

Testing that a weak reference object is still live should be done using the expression `ref.get() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```python
o = ref.get()
if o is None:
    # referent has been garbage collected
    print "Object has been allocated; can’t frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

Using a separate test for “liveness” creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the `get()` method is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

3.3.2 Example

This simple example shows how an application can use objects IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.
import weakref

__id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    __id2obj_dict[id(obj)] = obj

def id2obj(id):
    return __id2obj_dict.get(id)

3.3.3 Weak References in Extension Types

One of the goals of the implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on those objects which do not benefit by weak referencing (such as numbers).

For an object to be weakly referencable, the extension must include a PyObject * field in the instance structure for the use of the weak reference mechanism; it must be initialized to NULL by the object’s constructor. It must also set the tp_weaklistoffset field of the corresponding type object to the offset of the field. For example, the instance type is defined with the following structure:

```c
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class; /* The class object */
    PyObject *in_dict; /* A dictionary */
    PyObject *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

The statically-declared type object for instances is defined this way:

```c
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "instance",
    /* lots of stuff omitted for brevity */
    offsetof(PyInstanceObject, in_weakreflist) /* tp_weaklistoffset */
};
```

The only further addition is that the destructor needs to call the weak reference manager to clear any weak references and return if the object has been resurrected. This needs to occur before any other parts of the destruction have occurred:
static void
instance_dealloc(PyInstanceObject *inst)
{
    /* allocate temporaries if needed, but do not begin
destruction here */
    
    if (!PyObject_ClearWeakRefs((PyObject *) inst))
        return;
    
    /* proceed with object destruction normally */
}

3.4 fpectl — Floating point exception control

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real
computer, some floating point operations produce results that cannot be expressed as a normal floating point value.
For example, try

```python
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000)/math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) ”Inf” is a special, non-numeric
value in IEEE-754 that stands for ”infinity”, and ”nan” means ”not a number.” Note that, other than the non-numeric
results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default
behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty
operation was attempted. The fpectl module is for use in that situation. It provides control over floating point
units from several hardware manufacturers, allowing the user to turn on the generation of SIGFPE whenever any of
the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper
macros that are inserted into the C code comprising your python system, SIGFPE is trapped and converted into the
Python FloatingPointError exception.

The fpectl module defines the following functions and may raise the given exception:

- **turnon_sigfpe()**
  Turn on the generation of SIGFPE, and set up an appropriate signal handler.

- **turnoff_sigfpe()**
  Reset default handling of floating point exceptions.

- **exception FloatingPointError**
  After turnon_sigfpe() has been executed, a floating point operation that raises one of the IEEE-754 ex-
  ceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.
3.4.1 Example

The following example demonstrates how to start up and test operation of the fpectl module.

```python
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow PASS
FloatingPointError: Overflow
div by 0 PASS
FloatingPointError: Division by zero
[ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
FloatingPointError: in math_1
```

3.4.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify fpectl to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros `PyFPE_START_PROTECT` and `PyFPE_END_PROTECT` be inserted into your code in an appropriate fashion. Python itself has been modified to support the fpectl module, but many other codes of interest to numerical analysts have not.

The fpectl module is not thread-safe.

See Also:

Some files in the source distribution may be interesting in learning more about how this module operates. The include file `Include/pyfpe.h` discusses the implementation of this module at some length. `Modules/fpetestmodule.c` gives several examples of use. Many additional examples can be found in `Objects/floatobject.c`.

3.5 atexit — Exit handlers

New in version 2.0.

The atexit module defines a single function to register cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination.

Note: the functions registered via this module are not called when the program is killed by a signal, when a Python fatal internal error is detected, or when `os._exit()` is called.

This is an alternate interface to the functionality provided by the sys.exitfunc variable.

Note: This module is unlikely to work correctly when used with other code that sets `sys.exitfunc`. In particular, other core Python modules are free to use atexit without the programmer’s knowledge. Authors who use `sys.exitfunc` should convert their code to use atexit instead. The simplest way to convert code that sets `sys.exitfunc` is to import atexit and register the function that had been bound to `sys.exitfunc`.

```python
register(func[, *args[, **kargs ]])
```
Register `func` as a function to be executed at termination. Any optional arguments that are to be passed to `func` must be passed as arguments to `register()`.

At normal program termination (for instance, if `sys.exit()` is called or the main module’s execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

See Also:

Module `readline` (section 7.16):

Useful example of `atexit` to read and write `readline` history files.

### 3.5.1 atexit Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter’s updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```python
try:
    _count = int(open("/tmp/counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("/tmp/counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

### 3.6 types — Names for all built-in types

This module defines names for all object types that are used by the standard Python interpreter, but not for the types defined by various extension modules. It is safe to use `from types import *` — the module does not export any names besides the ones listed here. New names exported by future versions of this module will all end in ‘Type’.

Typical use is for functions that do different things depending on their argument types, like the following:

```python
from types import *

def delete(list, item):
    if type(item) is IntType:
        del list[item]
    else:
        list.remove(item)
```

The module defines the following names:

**NoneType**

The type of None.
**TypeType**
The type of type objects (such as returned by `type()`).

**IntType**
The type of integers (e.g. 1).

**LongType**
The type of long integers (e.g. 1L).

**FloatType**
The type of floating point numbers (e.g. 1.0).

**ComplexType**
The type of complex numbers (e.g. 1.0j).

**StringType**
The type of character strings (e.g. 'Spam').

**UnicodeType**
The type of Unicode character strings (e.g. u'Spam').

**TupleType**
The type of tuples (e.g. (1, 2, 3, 'Spam')).

**ListType**
The type of lists (e.g. [0, 1, 2, 3]).

**DictType**
The type of dictionaries (e.g. {'Bacon': 1, 'Ham': 0}).

**DictionaryType**
An alternate name for DictType.

**FunctionType**
The type of user-defined functions and lambdas.

**LambdaType**
An alternate name for FunctionType.

**CodeType**
The type for code objects such as returned by `compile()`.

**ClassType**
The type of user-defined classes.

**InstanceType**
The type of instances of user-defined classes.

**MethodType**
The type of methods of user-defined class instances.

**UnboundMethodType**
An alternate name for MethodType.

**BuiltinFunctionType**
The type of built-in functions like `len()` or `sys.exit()`.

**BuiltinMethodType**
An alternate name for BuiltinFunction.

**ModuleType**
The type of modules.

**FileType**
The type of open file objects such as `sys.stdout`. 
The type of range objects returned by `xrange()`.

The type of objects returned by `slice()`.

The type of `Ellipsis`.

The type of traceback objects such as found in `sys.exc_traceback`.

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

The type of buffer objects created by the `buffer()` function.

### 3.7 UserDict — Class wrapper for dictionary objects

This module defines a class that acts as a wrapper around dictionary objects. It is a useful base class for your own dictionary-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to dictionaries.

The `UserDict` module defines the `UserDict` class:

```python
class UserDict([initialdata])
```

Class that simulates a dictionary. The instance’s contents are kept in a regular dictionary, which is accessible via the `data` attribute of `UserDict` instances. If `initialdata` is provided, `data` is initialized with its contents; note that a reference to `initialdata` will not be kept, allowing it be used for other purposes.

In addition to supporting the methods and operations of mappings (see section 2.1.6), `UserDict` instances provide the following attribute:

```python
data
```

A real dictionary used to store the contents of the `UserDict` class.

### 3.8 UserList — Class wrapper for list objects

This module defines a class that acts as a wrapper around list objects. It is a useful base class for your own list-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to lists.

The `UserList` module defines the `UserList` class:

```python
class UserList([list])
```

Class that simulates a list. The instance’s contents are kept in a regular list, which is accessible via the `data` attribute of `UserList` instances. The instance’s contents are initially set to a copy of `list`, defaulting to the empty list `[]`. `list` can be either a regular Python list, or an instance of `UserList` (or a subclass).

In addition to supporting the methods and operations of mutable sequences (see section 2.1.5), `UserList` instances provide the following attribute:

```python
data
```

A real Python list object used to store the contents of the `UserList` class.

**Subclassing requirements:** Subclasses of `UserList` are expect to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the
actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

Changed in version 2.0: Python versions 1.5.2 and 1.6 also required that the constructor be callable with no parameters, and offer a mutable data attribute. Earlier versions of Python did not attempt to create instances of the derived class.

### 3.9 UserString — Class wrapper for string objects

This module defines a class that acts as a wrapper around string objects. It is a useful base class for your own string-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to strings.

It should be noted that these classes are highly inefficient compared to real string or Unicode objects; this is especially the case for MutableString.

The UserString module defines the following classes:

**class UserString ([sequence])**

Class that simulates a string or a Unicode string object. The instance’s content is kept in a regular string or Unicode string object, which is accessible via the data attribute of UserString instances. The instance’s contents are initially set to a copy of sequence. sequence can be either a regular Python string or Unicode string, an instance of UserString (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in str() function.

**class MutableString ([sequence])**

This class is derived from the UserString above and redefines strings to be mutable. Mutable strings can’t be used as dictionary keys, because dictionaries require immutable objects as keys. The main intention of this class is to serve as an educational example for inheritance and necessity to remove (override) the __hash__() method in order to trap attempts to use a mutable object as dictionary key, which would be otherwise very error prone and hard to track down.

In addition to supporting the methods and operations of string and Unicode objects (see section 2.1.5, “String Methods”), UserString instances provide the following attribute:

**data**

A real Python string or Unicode object used to store the content of the UserString class.

### 3.10 operator — Standard operators as functions.

The operator module exports a set of functions implemented in C corresponding to the intrinsic operators of Python. For example, operator.add(x, y) is equivalent to the expression x+y. The function names are those used for special class methods; variants without leading and trailing ‘__’ are also provided for convenience.

The operator module defines the following functions:

add (a, b)
__add__(a, b)
        Return a + b, for a and b numbers.

sub (a, b)
__sub__(a, b)
        Return a – b.
mul (a, b)
  __mul__(a, b)
  Return a * b, for a and b numbers.

div (a, b)
  __div__(a, b)
  Return a / b.

mod (a, b)
  __mod__(a, b)
  Return a % b.

neg (o)
  __neg__(o)
  Return o negated.
pos (o)
  __pos__(o)
  Return o positive.
abs (o)
  __abs__(o)
  Return the absolute value of o.
inv (o)
invert (o)
  __inv__(o)
  __invert__(o)
  Return the bitwise inverse of the number o. The names invert () and __invert__ () were added in Python 2.0.
lshift (a, b)
  __lshift__(a, b)
  Return a shifted left by b.
rshift (a, b)
  __rshift__(a, b)
  Return a shifted right by b.
and_ (a, b)
  __and__(a, b)
  Return the bitwise and of a and b.
or_ (a, b)
  __or__(a, b)
  Return the bitwise or of a and b.
xor (a, b)
  __xor__(a, b)
  Return the bitwise exclusive or of a and b.
not_ (o)
  __not__(o)
  Return the outcome of not o. (Note that there is no __not__ () method for object instances; only the interpreter core defines this operation.)
truth (o)
  Return 1 if o is true, and 0 otherwise.
concat (a, b)
  __concat__(a, b)
Return $a + b$ for $a$ and $b$ sequences.

**repeat** $(a, b)$

___repeat___$(a, b)$

Return $a \times b$ where $a$ is a sequence and $b$ is an integer.

**contains** $(a, b)$

___contains___$(a, b)$

Return the outcome of the test $b$ in $a$. Note the reversed operands. The name ___contains___() was added in Python 2.0.

**sequenceIncludes**(...)

Deprecated since release 2.0. Use contains() instead.

Alias for contains().

**countOf** $(a, b)$

Return the number of occurrences of $b$ in $a$.

**indexOf** $(a, b)$

Return the index of the first of occurrence of $b$ in $a$.

**getitem** $(a, b)$

___getitem___$(a, b)$

Return the value of $a$ at index $b$.

**setitem** $(a, b, c)$

___setitem___$(a, b, c)$

Set the value of $a$ at index $b$ to $c$.

**delitem** $(a, b)$

___delitem___$(a, b)$

Remove the value of $a$ at index $b$.

**getslice** $(a, b, c)$

___getslice___$(a, b, c)$

Return the slice of $a$ from index $b$ to index $c-1$.

**setslice** $(a, b, c, v)$

___setslice___$(a, b, c, v)$

Set the slice of $a$ from index $b$ to index $c-1$ to the sequence $v$.

**delslice** $(a, b, c)$

___delslice___$(a, b, c)$

Delete the slice of $a$ from index $b$ to index $c-1$.

The operator also defines a few predicates to test the type of objects. **Note:** Be careful not to misinterpret the results of these functions; only isCallable() has any measure of reliability with instance objects. For example:

```python
>>> class C:
...     pass
...     ...
>>> import operator
>>> o = C()
>>> operator.isMappingType(o)
1
```

**isCallable**$(o)$

Deprecated since release 2.0. Use the callable() built-in function instead.

Returns true if the object $o$ can be called like a function, otherwise it returns false. True is returned for functions,
bound and unbound methods, class objects, and instance objects which support the \_\_call\_\_() method.

**isMappingType(o)**

Returns true if the object \( o \) supports the mapping interface. This is true for dictionaries and all instance objects. **Warning:** There is no reliable way to test if an instance supports the complete mapping protocol since the interface itself is ill-defined. This makes this test less useful than it otherwise might be.

**isNumberType(o)**

Returns true if the object \( o \) represents a number. This is true for all numeric types implemented in C, and for all instance objects. **Warning:** There is no reliable way to test if an instance supports the complete numeric interface since the interface itself is ill-defined. This makes this test less useful than it otherwise might be.

**isSequenceType(o)**

Returns true if the object \( o \) supports the sequence protocol. This returns true for all objects which define sequence methods in C, and for all instance objects. **Warning:** There is no reliable way to test if an instance supports the complete sequence interface since the interface itself is ill-defined. This makes this test less useful than it otherwise might be.

Example: Build a dictionary that maps the ordinals from 0 to 256 to their character equivalents.

```python
>>> import operator
>>> d = {}
>>> keys = range(256)
>>> vals = map(chr, keys)
>>> map(operator.setitem, [d]*len(keys), keys, vals)
```

### 3.10.1 Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the `operator` module.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>$a + b$</td>
<td>$\text{add} (a, b)$</td>
</tr>
<tr>
<td>Concatenation</td>
<td>$\text{seq1} + \text{seq2}$</td>
<td>$\text{concat} (\text{seq1}, \text{seq2})$</td>
</tr>
<tr>
<td>Containment Test</td>
<td>$o \text{ in seq}$</td>
<td>$\text{contains} (\text{seq}, o)$</td>
</tr>
<tr>
<td>Division</td>
<td>$a / b$</td>
<td>$\text{div} (a, b)$</td>
</tr>
<tr>
<td>Bitwise And</td>
<td>$a &amp; b$</td>
<td>$\text{and} (a, b)$</td>
</tr>
<tr>
<td>Bitwise Exclusive Or</td>
<td>$a ^ b$</td>
<td>$\text{xor} (a, b)$</td>
</tr>
<tr>
<td>Bitwise Inversion</td>
<td>$\sim a$</td>
<td>$\text{invert} (a)$</td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>$a</td>
<td>b$</td>
</tr>
<tr>
<td>Indexed Assignment</td>
<td>$o [k] = v$</td>
<td>$\text{setitem} (o, k, v)$</td>
</tr>
<tr>
<td>Indexed Deletion</td>
<td>$\text{del} o [k]$</td>
<td>$\text{delitem} (o, k)$</td>
</tr>
<tr>
<td>Indexing</td>
<td>$o [k]$</td>
<td>$\text{getitem} (o, k)$</td>
</tr>
<tr>
<td>Left Shift</td>
<td>$a &lt;&lt; b$</td>
<td>$\text{lshift} (a, b)$</td>
</tr>
<tr>
<td>Modulo</td>
<td>$a % b$</td>
<td>$\text{mod} (a, b)$</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$a * b$</td>
<td>$\text{mul} (a, b)$</td>
</tr>
<tr>
<td>Negation (Arithmetic)</td>
<td>$- a$</td>
<td>$\text{neg} (a)$</td>
</tr>
<tr>
<td>Negation (Logical)</td>
<td>$\text{not} a$</td>
<td>$\text{not} (a)$</td>
</tr>
<tr>
<td>Right Shift</td>
<td>$a &gt;&gt; b$</td>
<td>$\text{rshift} (a, b)$</td>
</tr>
<tr>
<td>Sequence Repetition</td>
<td>$\text{seq} * i$</td>
<td>$\text{repeat} (\text{seq}, i)$</td>
</tr>
<tr>
<td>Slice Assignment</td>
<td>$\text{seq[i:j]} = \text{values}$</td>
<td>$\text{setslice} (\text{seq}, i, j, \text{values})$</td>
</tr>
<tr>
<td>Slice Deletion</td>
<td>$\text{del} \text{seq[i:j]}$</td>
<td>$\text{delslice} (\text{seq}, i, j)$</td>
</tr>
<tr>
<td>Slicing</td>
<td>$\text{seq[i:j]}$</td>
<td>$\text{getslice} (\text{seq}, i, j)$</td>
</tr>
<tr>
<td>String Formatting</td>
<td>$s % o$</td>
<td>$\text{mod} (s, o)$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$a - b$</td>
<td>$\text{sub} (a, b)$</td>
</tr>
<tr>
<td>Truth Test</td>
<td>$o$</td>
<td>$\text{truth} (o)$</td>
</tr>
</tbody>
</table>

### 3.11 inspect — Inspect live objects

New in version 2.1.

The `inspect` module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

#### 3.11.1 Types and members

The `getmembers()` function retrieves the members of an object such as a class or module. The nine functions whose names begin with “is” are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes:
<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>file</strong></td>
<td>filename (missing for built-in modules)</td>
</tr>
<tr>
<td>class</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>module</strong></td>
<td>name of module in which this class was defined</td>
</tr>
<tr>
<td>method</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this method was defined</td>
</tr>
<tr>
<td></td>
<td>im_class</td>
<td>class object in which this method belongs</td>
</tr>
<tr>
<td></td>
<td>im_func</td>
<td>function object containing implementation of method</td>
</tr>
<tr>
<td></td>
<td>im_self</td>
<td>instance to which this method is bound, or None</td>
</tr>
<tr>
<td>function</td>
<td><strong>doc</strong></td>
<td>documentation string</td>
</tr>
<tr>
<td></td>
<td><strong>name</strong></td>
<td>name with which this function was defined</td>
</tr>
<tr>
<td></td>
<td>func_code</td>
<td>code object containing compiled function bytecode</td>
</tr>
<tr>
<td></td>
<td>func_defaults</td>
<td>tuple of any default values for arguments</td>
</tr>
<tr>
<td></td>
<td>func_doc</td>
<td>(same as <strong>doc</strong>)</td>
</tr>
<tr>
<td></td>
<td>func_globals</td>
<td>global namespace in which this function was defined</td>
</tr>
<tr>
<td></td>
<td>func_name</td>
<td>(same as <strong>name</strong>)</td>
</tr>
<tr>
<td>traceback</td>
<td>tb_frame</td>
<td>frame object at this level</td>
</tr>
<tr>
<td></td>
<td>tb_lasti</td>
<td>index of last attempted instruction in bytecode</td>
</tr>
<tr>
<td></td>
<td>tb_lineno</td>
<td>current line number in Python source code</td>
</tr>
<tr>
<td></td>
<td>tb_next</td>
<td>next inner traceback object (called by this level)</td>
</tr>
<tr>
<td>frame</td>
<td>f_back</td>
<td>next outer frame object (this frame’s caller)</td>
</tr>
<tr>
<td></td>
<td>f_builtins</td>
<td>built-in namespace seen by this frame</td>
</tr>
<tr>
<td></td>
<td>f_code</td>
<td>code object being executed in this frame</td>
</tr>
<tr>
<td></td>
<td>f_exc_traceback</td>
<td>traceback if raised in this frame, or None</td>
</tr>
<tr>
<td></td>
<td>f_exc_type</td>
<td>exception type if raised in this frame, or None</td>
</tr>
<tr>
<td></td>
<td>f_exc_value</td>
<td>exception value if raised in this frame, or None</td>
</tr>
<tr>
<td></td>
<td>f_globals</td>
<td>global namespace seen by this frame</td>
</tr>
<tr>
<td></td>
<td>f_lasti</td>
<td>index of last attempted instruction in bytecode</td>
</tr>
<tr>
<td></td>
<td>f_lineno</td>
<td>current line number in Python source code</td>
</tr>
<tr>
<td></td>
<td>f_locals</td>
<td>local namespace seen by this frame</td>
</tr>
<tr>
<td></td>
<td>f_restricted</td>
<td>0 or 1 if frame is in restricted execution mode</td>
</tr>
<tr>
<td></td>
<td>f_trace</td>
<td>tracing function for this frame, or None</td>
</tr>
<tr>
<td>code</td>
<td>co_argcount</td>
<td>number of arguments (not including * or ** args)</td>
</tr>
<tr>
<td></td>
<td>co_code</td>
<td>string of raw compiled bytecode</td>
</tr>
<tr>
<td></td>
<td>co_consts</td>
<td>tuple of constants used in the bytecode</td>
</tr>
<tr>
<td></td>
<td>co_filename</td>
<td>name of file in which this code object was created</td>
</tr>
<tr>
<td></td>
<td>co_filename</td>
<td>number of first line in Python source code</td>
</tr>
<tr>
<td></td>
<td>co_flags</td>
<td>bitmap: 1=optimized</td>
</tr>
<tr>
<td></td>
<td>co_inotab</td>
<td>encoded mapping of line numbers to bytecode indices</td>
</tr>
<tr>
<td></td>
<td>co_name</td>
<td>name with which this code object was defined</td>
</tr>
<tr>
<td></td>
<td>co_names</td>
<td>tuple of names of local variables</td>
</tr>
<tr>
<td></td>
<td>co_nlocals</td>
<td>number of local variables</td>
</tr>
<tr>
<td></td>
<td>co_stacksize</td>
<td>virtual machine stack space required</td>
</tr>
<tr>
<td></td>
<td>co_varnames</td>
<td>tuple of names of arguments and local variables</td>
</tr>
</tbody>
</table>

**getmembers**(object[, predicate])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional predicate argument is supplied, only members for which the predicate returns a true value are included.

**getmoduleinfo**(path)

Return a tuple of values that describe how Python will interpret the file identified by path if it is a module, or
None if it would not be identified as a module. The return tuple is \((\text{name}, \text{suffix}, \text{mode}, \text{mtype})\), where
\text{name} is the name of the module without the name of any enclosing package, \text{suffix} is the trailing part of the file
name (which may not be a dot-delimited extension), \text{mode} is the \text{open()} mode that would be used (‘r’ or
‘rb’), and \text{mtype} is an integer giving the type of the module. \text{mtype} will have a value which can be compared
to the constants defined in the \text{imp} module; see the documentation for that module for more information on
module types.

\text{getmodulename}(\text{path})
Return the name of the module named by the file \text{path}, without including the names of enclosing packages. This
uses the same algorithm as the interpreter uses when searching for modules. If the name cannot be matched
according to the interpreter’s rules, \text{None} is returned.

\text{ismodule}(\text{object})
Return true if the object is a module.

\text{isclass}(\text{object})
Return true if the object is a class.

\text{ismethod}(\text{object})
Return true if the object is a method.

\text{isfunction}(\text{object})
Return true if the object is a Python function or unnamed (lambda) function.

\text{istraceback}(\text{object})
Return true if the object is a traceback.

\text{isframe}(\text{object})
Return true if the object is a frame.

\text{iscode}(\text{object})
Return true if the object is a code.

\text{isbuiltin}(\text{object})
Return true if the object is a built-in function.

\text{isroutine}(\text{object})
Return true if the object is a user-defined or built-in function or method.

3.11.2 Retrieving source code

\text{getdoc}(\text{object})
Get the documentation string for an object. All tabs are expanded to spaces. To clean up docstrings that are
indented to line up with blocks of code, any whitespace than can be uniformly removed from the second line
onwards is removed.

\text{getcomments}(\text{object})
Return in a single string any lines of comments immediately preceding the object’s source code (for a class,
function, or method), or at the top of the Python source file (if the object is a module).

\text{getfile}(\text{object})
Return the name of the (text or binary) file in which an object was defined. This will fail with a \text{TypeError}
if the object is a built-in module, class, or function.

\text{getmodule}(\text{object})
Try to guess which module an object was defined in.

\text{getsourcefile}(\text{object})
Return the name of the Python source file in which an object was defined. This will fail with a \text{TypeError}
if the object is a built-in module, class, or function.
`getsource` *(object)*

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `IOError` is raised if the source code cannot be retrieved.

### 3.11.3 Classes and functions

`getclasstree` *(classes[, unique]*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the `unique` argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`getargspec` *(func)*

Get the names and default values of a function’s arguments. A tuple of four things is returned: *(args, varargs, varkw, defaults)*. `args` is a list of the argument names (it may contain nested lists). `varargs` and `varkw` are the names of the * and ** arguments or `None`. `defaults` is a tuple of default argument values; if this tuple has `n` elements, they correspond to the last `n` elements listed in `args`.

`getargvalues` *(frame)*

Get information about arguments passed into a particular frame. A tuple of four things is returned: *(args, varargs, varkw, locals)*. `args` is a list of the argument names (it may contain nested lists). `varargs` and `varkw` are the names of the * and ** arguments or `None`. `locals` is the locals dictionary of the given frame.

`formatargspec` *(args[, varargs, varkw, defaults, argformat, varargsformat, varkwformat, defaultformat]*)

Format a pretty argument spec from the four values returned by `getargspec()` . The other four arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

`formatargvalues` *(args[, varargs, varkw, locals, argformat, varargsformat, varkwformat, valueformat]*)

Format a pretty argument spec from the four values returned by `getargvalues()` . The other four arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

### 3.11.4 The interpreter stack

When the following functions return “frame records,” each record is a tuple of six items: the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list. The optional `context` argument specifies the number of lines of context to return, which are centered around the current line.

`getouterframes` *(frame[, context]*)

Get a list of frame records for a frame and all higher (calling) frames.

`getinnerframes` *(traceback[, context]*)

Get a list of frame records for a traceback’s frame and all lower frames.

`currentframe` *

Return the frame object for the caller’s stack frame.

`stack` *(context)*

Return a list of frame records for the stack above the caller’s frame.
trace([context])
   Return a list of frame records for the stack below the current exception.

3.12 traceback — Print or retrieve a stack traceback

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, e.g. in a “wrapper” around the interpreter.

The module uses traceback objects — this is the object type that is stored in the variables sys.exc_traceback and sys.last_traceback and returned as the third item from sys.exc_info().

The module defines the following functions:

print_tb(traceback[, limit[, file]])
   Print up to limit stack trace entries from traceback. If limit is omitted or None, all entries are printed. If file is omitted or None, the output goes to sys.stderr; otherwise it should be an open file or file-like object to receive the output.

print_exception(type, value, traceback[, limit[, file]])
   Print exception information and up to limit stack trace entries from traceback to file. This differs from print_tb() in the following ways: (1) if traceback is not None, it prints a header 'Traceback (most recent call last):'; (2) it prints the exception type and value after the stack trace; (3) if type is SyntaxError and value has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

print_exc([limit[, file]])
   This is a shorthand for print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file). (In fact, it uses sys.exc_info() to retrieve the same information in a thread-safe way.)

print_last([limit[, file]])
   This is a shorthand for print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file).

print_stack([f[, limit[, file]]])
   This function prints a stack trace from its invocation point. The optional f argument can be used to specify an alternate stack frame to start. The optional limit and file arguments have the same meaning as for print_exception().

extract_tb(traceback[, limit])
   Return a list of up to limit “pre-processed” stack trace entries extracted from the traceback object traceback. It is useful for alternate formatting of stack traces. If limit is omitted or None, all entries are extracted. A “pre-processed” stack trace entry is a quadruple (filename, line number, function name, text) representing the information that is usually printed for a stack trace. The text is a string with leading and trailing whitespace stripped; if the source is not available it is None.

extract_stack([f[, limit]])
   Extract the raw traceback from the current stack frame. The return value has the same format as for extract_tb(). The optional f and limit arguments have the same meaning as for print_stack().

format_list(list)
   Given a list of tuples as returned by extract_tb() or extract_stack(), return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not None.

format_exception_only(type, value)
Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

```python
format_exception(type, value, tb[, limit])
```

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.

```python
format_tb(tb[, limit])
```

A shorthand for `format_list(extract_tb(tb, limit))`.

```python
format_stack([f[, limit]])
```

A shorthand for `format_list(extract_stack(f, limit))`.

```python
tb_lineno(tb)
```

This function returns the current line number set in the traceback object. This is normally the same as the `tb.tb_lineno` field of the object, but when optimization is used (the `-O` flag) this field is not updated correctly; this function calculates the correct value.

### 3.12.1 Traceback Example

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the `code` module.

```python
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print ' ' * 60
        traceback.print_exc(file=sys.stdout)
        print ' ' * 60

envdir = {}
while 1:
    run_user_code(envdir)
```

### 3.13 linecache — Random access to text lines

The `linecache` module allows one to get any line from any file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the `traceback` module to retrieve source lines for inclusion in the formatted traceback.

The `linecache` module defines the following functions:

```python
getline(filename, lineno)
```

Get line `lineno` from file named `filename`. This function will never throw an exception — it will return `'` on
errors (the terminating newline character will be included for lines that are found).

If a file named filename is not found, the function will look for it in the module search path, sys.path.

```
clearcache()
```
Clear the cache. Use this function if you no longer need lines from files previously read using getline().

```
checkcache()
```
Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version.

Example:

```
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh
'
```

3.14 pickle — Python object serialization

The pickle module implements a basic but powerful algorithm for “pickling” (a.k.a. serializing, marshalling or flattening) nearly arbitrary Python objects. This is the act of converting objects to a stream of bytes (and back: “unpickling”). This is a more primitive notion than persistence — although pickle reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) area of concurrent access to persistent objects. The pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. The most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module shelve provides a simple interface to pickle and unpickle objects on DBM-style database files.

**Note:** The pickle module is rather slow. A reimplementation of the same algorithm in C, which is up to 1000 times faster, is available as the cPickle module. This has the same interface except that Pickler and Unpickler are factory functions, not classes (so they cannot be used as base classes for inheritance).

Although the pickle module can use the built-in module marshal internally, it differs from marshal in the way it handles certain kinds of data:

- Recursive objects (objects containing references to themselves): pickle keeps track of the objects it has already serialized, so later references to the same object won’t be serialized again. (The marshal module breaks for this.)

- Object sharing (references to the same object in different places): This is similar to self-referencing objects; pickle stores the object once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- User-defined classes and their instances: marshal does not support these at all, but pickle can save and restore class instances transparently. The class definition must be importable and live in the same module as when the object was stored.

The data format used by pickle is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as XDR (which can’t represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the pickle data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. The big advantage of using printable ASCII (and of some other characteristics of pickle’s representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor.
A binary format, which is slightly more efficient, can be chosen by specifying a nonzero (true) value for the `bin` argument to the `Pickler` constructor or the `dump()` and `dumps()` functions. The binary format is not the default because of backwards compatibility with the Python 1.4 pickle module. In a future version, the default may change to binary.

The `pickle` module doesn’t handle code objects, which the `marshal` module does. I suppose `pickle` could, and maybe it should, but there’s probably no great need for it right now (as long as `marshal` continues to be used for reading and writing code objects), and at least this avoids the possibility of smuggling Trojan horses into a program.

For the benefit of persistence modules written using `pickle`, it supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a name, which is an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the `pickle` module — the persistent object module will have to implement a method `persistent_load()`. To write references to persistent objects, the persistent module must define a method `persistent_id()` which returns either `None` or the persistent ID of the object.

There are some restrictions on the pickling of class instances.

First of all, the class must be defined at the top level in a module. Furthermore, all its instance variables must be picklable.

When a pickled class instance is unpickled, its `__init__()` method is normally not invoked. **Note:** This is a deviation from previous versions of this module; the change was introduced in Python 1.5b2. The reason for the change is that in many cases it is desirable to have a constructor that requires arguments; it is a (minor) nuisance to have to provide a `__getinitargs__()` method.

If it is desirable that the `__init__()` method be called on unpickling, a class can define a method `__getinitargs__()`, which should return a tuple containing the arguments to be passed to the class constructor (`__init__()`). This method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

Classes can further influence how their instances are pickled — if the class defines the method `__getstate__()`, it is called and the return state is pickled as the contents for the instance, and if the class defines the method `__setstate__()`, it is called with the unpickled state. (Note that these methods can also be used to implement copying class instances.) If there is no `__getstate__()` method, the instance’s `__dict__` is pickled. If there is no `__setstate__()` method, the pickled object must be a dictionary and its items are assigned to the new instance’s dictionary. (If a class defines both `__getstate__()` and `__setstate__()`, the state object needn’t be a dictionary — these methods can do what they want.) This protocol is also used by the shallow and deep copying operations defined in the `copy` module.

Note that when class instances are pickled, their class’s code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class’s `__setstate__()` method.

When a class itself is pickled, only its name is pickled — the class definition is not pickled, but re-imported by the unpickling process. Therefore, the restriction that the class must be defined at the top level in a module applies to pickled classes as well.

The interface can be summarized as follows.

To pickle an object `x` onto a file `f`, open for writing:

```python
p = pickle.Pickler(f)
p.dump(x)
```

A shorthand for this is:

```python
pickle.dump(x, f)
```

3.14. `pickle` — Python object serialization
To unpickle an object \( x \) from a file \( f \), open for reading:

\[
\begin{align*}
u &= \text{pickle.Unpickler}(f) \\
x &= u.load()
\end{align*}
\]

A shorthand is:

\[
x = \text{pickle.load}(f)
\]

The \texttt{Pickler} class only calls the method \( f.write() \) with a string argument. The \texttt{Unpickler} calls the methods \( f.read() \) (with an integer argument) and \( f.readline() \) (without argument), both returning a string. It is explicitly allowed to pass non-file objects here, as long as they have the right methods.

The constructor for the \texttt{Pickler} class has an optional second argument, \texttt{bin}. If this is present and true, the binary pickle format is used; if it is absent or false, the (less efficient, but backwards compatible) text pickle format is used. The \texttt{Unpickler} class does not have an argument to distinguish between binary and text pickle formats; it accepts either format.

The following types can be pickled:

- \texttt{None}
- \texttt{integers}, long integers, floating point numbers
- \texttt{normal} and \texttt{Unicode} strings
- \texttt{tuples}, lists and dictionaries containing only picklable objects
- \texttt{functions} defined at the top level of a module (by name reference, not storage of the implementation)
- \texttt{built-in} functions
- \texttt{classes} that are defined at the top level in a module
- \texttt{instances} of such classes whose \texttt{__dict__} or \texttt{__setstate__() is picklable}

Attempts to pickle unpicklable objects will raise the \texttt{PicklingError} exception; when this happens, an unspecified number of bytes may have been written to the file.

It is possible to make multiple calls to the \texttt{dump()} method of the same \texttt{Pickler} instance. These must then be matched to the same number of calls to the \texttt{load()} method of the corresponding \texttt{Unpickler} instance. If the same object is pickled by multiple \texttt{dump()} calls, the \texttt{load()} will all yield references to the same object. \textit{Warning}: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same \texttt{Pickler} instance, the object is not pickled again — a reference to it is pickled and the \texttt{Unpickler} will return the old value, not the modified one. (There are two problems here: (a) detecting changes, and (b) marshalling a minimal set of changes. I have no answers. Garbage Collection may also become a problem here.)

Apart from the \texttt{Pickler} and \texttt{Unpickler} classes, the module defines the following functions, and an exception:

\textbf{dump(object, file[, bin])}

Write a pickled representation of \texttt{object} to the open file object \texttt{file}. This is equivalent to \texttt{'Pickler(file, bin).dump(object)'}. If the optional \texttt{bin} argument is present and nonzero, the binary pickle format is used; if it is zero or absent, the (less efficient) text pickle format is used.

\textbf{load(file)}

Read a pickled object from the open file object \texttt{file}. This is equivalent to \texttt{'Unpickler(file).load()'}.
`dumps(object[, bin])`

Return the pickled representation of the object as a string, instead of writing it to a file. If the optional `bin` argument is present and nonzero, the binary pickle format is used; if it is zero or absent, the (less efficient) text pickle format is used.

`loads(string)`

Read a pickled object from a string instead of a file. Characters in the string past the pickled object’s representation are ignored.

`exception PicklingError`

This exception is raised when an unpicklable object is passed to `Pickler.dump()`.

See Also:

- `Module copy_reg` (section 3.16): Pickle interface constructor registration for extension types.
- `Module shelve` (section 3.17): Indexed databases of objects; uses pickle.
- `Module copy` (section 3.18): Shallow and deep object copying.

### 3.14.1 Example

Here’s a simple example of how to modify pickling behavior for a class. The `TextReader` class opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a `TextReader` instance is pickled, all attributes except the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The `__setstate__()` and `__getstate__()` methods are used to implement this behavior.
# illustrate __setstate__ and __getstate__ methods
# used in pickling.

class TextReader:
    "Print and number lines in a text file."
    def __init__(self,file):
        self.file = file
        self.fh = open(file,'r')
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        return "%d: %s" % (self.lineno,line[:-1])

    # return data representation for pickled object
    def __getstate__(self):
        odict = self.__dict__ # get attribute dictionary
        del odict['fh'] # remove filehandle entry
        return odict

    # restore object state from data representation generated
    # by __getstate__
    def __setstate__(self,dict):
        fh = open(dict['file']) # reopen file
        count = dict['lineno'] # read from file...
        while count:
            fh.readline() # until line count is restored
            count = count - 1
        dict['fh'] = fh # create filehandle entry
        self.__dict__ = dict # make dict our attribute dictionary

A sample usage might be something like this:

    >>> import TextReader
    >>> obj = TextReader.TextReader("TextReader.py")
    >>> obj.readline()
    '1: #!/usr/local/bin/python'
    >>> # (more invocations of obj.readline() here)
    >>> obj.readline()
    '7: class TextReader:'
    >>> import pickle
    >>> pickle.dump(obj,open('save.p','w'))

    (start another Python session)

    >>> import pickle
    >>> reader = pickle.load(open('save.p'))
    >>> reader.readline()
    '8: "Print and number lines in a text file."'
### 3.15 cPickle — Alternate implementation of pickle

The cPickle module provides a similar interface and identical functionality as the pickle module, but can be up to 1000 times faster since it is implemented in C. The only other important difference to note is that Pickler() and Unpickler() are functions and not classes, and so cannot be subclassed. This should not be an issue in most cases.

The format of the pickle data is identical to that produced using the pickle module, so it is possible to use pickle and cPickle interchangeably with existing pickles.

(Since the pickle data format is actually a tiny stack-oriented programming language, and there are some freedoms in the encodings of certain objects, it’s possible that the two modules produce different pickled data for the same input objects; however they will always be able to read each other’s pickles back in.)

### 3.16 copy_reg — Register pickle support functions

The copy_reg module provides support for the pickle and cPickle modules. The copy module is likely to use this in the future as well. It provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

- **constructor**(object)
  
  Declares object to be a valid constructor. If object is not callable (and hence not valid as a constructor), raises TypeError.

- **pickle**(type, function[, constructor])
  
  Declares that function should be used as a “reduction” function for objects of type type; type should not a class object. function should return either a string or a tuple. The optional constructor parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by function at pickling time. TypeError will be raised if object is a class or constructor is not callable.

### 3.17 shelve — Python object persistence

A “shelf” is a persistent, dictionary-like object. The difference with “dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

To summarize the interface (key is a string, data is an arbitrary object):

```python
import shelve

d = shelve.open(filename) # open, with (g)dbm filename -- no suffix

d[key] = data      # store data at key (overwrites old data if
                 # using an existing key)
data = d[key]       # retrieve data at key (raise KeyError if no
                 # such key)
del d[key]         # delete data stored at key (raises KeyError
                 # if no such key)
flag = d.has_key(key)    # true if the key exists
list = d.keys()       # a list of all existing keys (slow!)

d.close()            # close it
```
Restrictions:

- The choice of which database package will be used (e.g. dbm or gdbm) depends on which interface is available. Therefore it is not safe to open the database directly using dbm. The database is also (unfortunately) subject to the limitations of dbm, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.

- Dependent on the implementation, closing a persistent dictionary may or may not be necessary to flush changes to disk.

- The shelf module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. UNIX file locking can be used to solve this, but this differs across UNIX versions and requires knowledge about the database implementation used.

See Also:

Module anydbm (section 7.8):
Generic interface to dbm-style databases.

Module dbhash (section 7.10):
BSD db database interface.

Module dbm (section 8.6):
Standard UNIX database interface.

Module dumbdbm (section 7.9):
Portable implementation of the dbm interface.

Module gdbm (section 8.7):
GNU database interface, based on the dbm interface.

Module pickle (section 3.14):
Object serialization used by shelf.

Module cPickle (section 3.15):
High-performance version of pickle.

3.18 copy — Shallow and deep copy operations

This module provides generic (shallow and deep) copying operations.

Interface summary:

```python
import copy

x = copy.copy(y)       # make a shallow copy of y
x = copy.deepcopy(y)   # make a deep copy of y
```

For module specific errors, copy.error is raised.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don’t exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, e.g., administrative data structures that should be shared even between copies.

The deepcopy() function avoids these problems by:

- keeping a “memo” dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This version does not copy types like module, class, function, method, stack trace, stack frame, file, socket, window, array, or any similar types.

Classes can use the same interfaces to control copying that they use to control pickling: they can define methods called __getinitargs__(), __getstate__() and __setstate__() and __deepcopy__() and __setstate__(). See the description of module pickle for information on these methods. The copy module does not use the copy_reg registration module.

In order for a class to define its own copy implementation, it can define special methods __copy__() and __deepcopy__(). The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the __deepcopy__() implementation needs to make a deep copy of a component, it should call the deepcopy() function with the component as first argument and the memo dictionary as second argument.

See Also:

Module pickle (section 3.14):
Discussion of the special methods used to support object state retrieval and restoration.

3.19 marshal — Alternate Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does).\(^1\)

This is not a general “persistence” module. For general persistence and transfer of Python objects through RPC calls, see the modules pickle and shelve. The marshal module exists mainly to support reading and writing the “pseudo-compiled” code for Python modules of `.pyc` files.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: None, integers, long integers, floating point numbers, strings, Unicode objects, tuples, lists, dictionaries, and code objects, where it should be understood that tuples, lists and dictionaries are only supported as long as the values contained therein are themselves supported; and recursive lists and dictionaries should not be written (they will cause infinite loops).

---

^1The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term “marshalling” for shipping of data around in a self-contained form. Strictly speaking, “to marshal” means to convert some data from internal to external form (in an RPC buffer for instance) and “unmarshalling” for the reverse process.
Caveat: On machines where C’s `long int` type has more than 32 bits (such as the DEC Alpha), it is possible to create plain Python integers that are longer than 32 bits. Since the current `marshal` module uses 32 bits to transfer plain Python integers, such values are silently truncated. This particularly affects the use of very long integer literals in Python modules — these will be accepted by the parser on such machines, but will be silently be truncated when the module is read from the `.pyc` instead.\(^2\)

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

- **dump**(value, file)
  - Write the value on the open file. The value must be a supported type. The file must be an open file object such as `sys.stdout` or returned by `open()` or `posix.popen()`. It must be opened in binary mode (`'wb'` or `'w+b'`).
  - If the value has (or contains an object that has) an unsupported type, a `ValueError` exception is raised — but garbage data will also be written to the file. The object will not be properly read back by `load()`.

- **load**(file)
  - Read one value from the open file and return it. If no valid value is read, raise `EOFError`, `ValueError` or `TypeError`. The file must be an open file object opened in binary mode (`'rb'` or `'r+b'`).
  - **Warning:** If an object containing an unsupported type was marshalled with `dump()`, `load()` will substitute None for the unmarshallable type.

- **dumps**(value)
  - Return the string that would be written to a file by `dump(value, file)`. The value must be a supported type.
  - Raise a `ValueError` exception if value has (or contains an object that has) an unsupported type.

- **loads**(string)
  - Convert the string to a value. If no valid value is found, raise `EOFError`, `ValueError` or `TypeError`. Extra characters in the string are ignored.

### 3.20 warnings — Warning control

New in version 2.1.

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn’t warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the `warn()` function defined in this module. (C programmers use `PyErr_Warn();` see the Python/C API Reference Manual for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling `resetwarnings()`.

\(^2\)A solution would be to refuse such literals in the parser, since they are inherently non-portable. Another solution would be to let the `marshal` module raise an exception when an integer value would be truncated. At least one of these solutions will be implemented in a future version.
The printing of warning messages is done by calling `showwarning()`, which may be overridden; the default implementation of this function formats the message by calling `formatwarning()`, which is also available for use by custom implementations.

### 3.20.1 Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>This is the base class of all warning category classes. It itself a subclass of Exception.</td>
</tr>
<tr>
<td>UserWarning</td>
<td>The default category for <code>warn()</code>.</td>
</tr>
<tr>
<td>DeprecationWarning</td>
<td>Base category for warnings about deprecated features.</td>
</tr>
<tr>
<td>SyntaxWarning</td>
<td>Base category for warnings about dubious syntactic features.</td>
</tr>
<tr>
<td>RuntimeWarning</td>
<td>Base category for warnings about dubious runtime features.</td>
</tr>
</tbody>
</table>

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the `Warning` class.

### 3.20.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form `(action, message, category, module, lineno)`, where:

- **action** is one of the following strings:
  - "error" turn matching warnings into exceptions
  - "ignore" never print matching warnings
  - "always" always print matching warnings
  - "default" print the first occurrence of matching warnings for each location where the warning is issued
  - "module" print the first occurrence of matching warnings for each module where the warning is issued
  - "once" print only the first occurrence of matching warnings, regardless of location

- **message** is a compiled regular expression that the warning message must match (the match is case-insensitive)

- **category** is a class (a subclass of `Warning`) of which the warning category must be a subclass in order to match

- **module** is a compiled regular expression that the module name must match

- **lineno** is an integer that the line number where the warning occurred must match, or 0 to match all line numbers

Since the `Warning` class is derived from the built-in `Exception` class, to turn a warning into an error we simply raise `category(message)`.

The warnings filter is initialized by `-W` options passed to the Python interpreter command line. The interpreter saves the arguments for all `-W` options without interpretation in `sys.warnoptions`; the `warnings` module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

---

3.20. `warnings` — Warning control 63
3.20.3 Available Functions

`warn (message[, category[, stacklevel ]])`
Issue a warning, or maybe ignore it or raise an exception. The `category` argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The `stacklevel` argument can be used by wrapper functions written in Python, like this:

```python
def deprecation(message):
    warnings.warn(message, DeprecationWarning, level=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

`warn_explicit (message, category, filename, lineno[, module[, registry ]])`
This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with .py stripped; if no registry is passed, the warning is never suppressed.

`showwarning (message, category, filename, lineno[, file ])`
Write a warning to a file. The default implementation calls `showwarning(message, category, filename, lineno)` and writes the resulting string to `file`, which defaults to `sys.stderr`. You may replace this function with an alternative implementation by assigning to `warnings.showwarning`.

`formatwarning (message, category, filename, lineno)`
Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline.

`filterwarnings (action[, message[, category[, module[, lineno[, append]]]]])`
Insert an entry into the list of warnings filters. The entry is inserted at the front by default; if `append` is true, it is inserted at the end. This checks the types of the arguments, compiles the message and module regular expressions, and inserts them as a tuple in front of the warnings filter. Entries inserted later override entries inserted earlier, if both match a particular warning. Omitted arguments default to a value that matches everything.

`resetwarnings ()`
Reset the warnings filter. This discards the effect of all previous calls to `filterwarnings ()`, including that of the `-W` command line options.

3.21 `imp` — Access the `import` internals

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

`get_magic ()`
Return the magic string value used to recognize byte-compiled code files (`*.pyc` files). (This value may be different for each Python version.)

`get_suffixes ()`
Return a list of triples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where `suffix` is a string to be appended to the module name to form the filename to search for, `mode` is the mode string to pass to the built-in `open()` function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and `type` is the file type, which has one of the values `PY_SOURCE`, `PY_COMPILED`, or `C_EXTENSION`, described below.

`find_module (name[, path ])`
Try to find the module `name` on the search path `path`. If `path` is a list of directory names, each directory is
searched for files with any of the suffixes returned by get_suffixes() above. Invalid names in the list are silently ignored (but all list items must be strings). If path is omitted or None, the list of directory names given by sys.path is searched, but first it searches a few special places: it tries to find a built-in module with the given name (C_BUILTIN), then a frozen module (PY_FROZEN), and on some systems some other places are looked in as well (on the Mac, it looks for a resource (PY_RESOURCE); on Windows, it looks in the registry which may point to a specific file).

If search is successful, the return value is a triple (file, pathname, description) where file is an open file object positioned at the beginning, pathname is the pathname of the file found, and description is a triple as contained in the list returned by get_suffixes() describing the kind of module found. If the module does not live in a file, the returned file is None, filename is the empty string, and the description tuple contains empty strings for its suffix and mode; the module type is as indicate in parentheses above. If the search is unsuccessful, ImportError is raised. Other exceptions indicate problems with the arguments or environment.

This function does not handle hierarchical module names (names containing dots). In order to find P.M, i.e., submodule M of package P, use find_module() and load_module() to find and load package P, and then use find_module() with the path argument set to P.__path__. When P itself has a dotted name, apply this recipe recursively.

**load_module**(name, file, filename, description)
Load a module that was previously found by find_module() (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it is equivalent to a reload()! The name argument indicates the full module name (including the package name, if this is a submodule of a package). The file argument is an open file, and filename is the corresponding file name; these can be None and '', respectively, when the module is not being loaded from a file. The description argument is a tuple, as would be returned by get_suffixes(), describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually ImportError) is raised.

**Important:** the caller is responsible for closing the file argument, if it was not None, even when an exception is raised. This is best done using a try ... finally statement.

**new_module**(name)
Return a new empty module object called name. This object is not inserted in sys.modules.

The following constants with integer values, defined in this module, are used to indicate the search result of find_module().

**PY_SOURCE**
The module was found as a source file.

**PY_COMPILED**
The module was found as a compiled code object file.

**C_EXTENSION**
The module was found as dynamically loadable shared library.

**PY_RESOURCE**
The module was found as a Macintosh resource. This value can only be returned on a Macintosh.

**PKG_DIRECTORY**
The module was found as a package directory.

**C_BUILTIN**
The module was found as a built-in module.

**PY_FROZEN**
The module was found as a frozen module (see init_frozen()).

The following constant and functions are obsolete; their functionality is available through find_module() or load_module(). They are kept around for backward compatibility:
**SEARCH_ERROR**

Unused.

**init_builtin**(name)

Initialize the built-in module called name and return its module object. If the module was already initialized, it will be initialized again. A few modules cannot be initialized twice — attempting to initialize these again will raise an `ImportError` exception. If there is no built-in module called name, None is returned.

**init_frozen**(name)

Initialize the frozen module called name and return its module object. If the module was already initialized, it will be initialized again. If there is no frozen module called name, None is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python’s `freeze` utility. See ’Tools/freeze/’ for now.)

**is_builtin**(name)

Return 1 if there is a built-in module called name which can be initialized again. Return -1 if there is a built-in module called name which cannot be initialized again (see `init_builtin()`). Return 0 if there is no built-in module called name.

**is_frozen**(name)

Return 1 if there is a frozen module (see `init_frozen()`) called name, or 0 if there is no such module.

**load_compiled**(name, pathname, file)

Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized again. The name argument is used to create or access a module object. The pathname argument points to the byte-compiled code file. The file argument is the byte-compiled code file, open for reading in binary mode, from the beginning. It must currently be a real file object, not a user-defined class emulating a file.

**load_dynamic**(name, pathname[, file])

Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized again. Some modules don’t like that and may raise an exception. The pathname argument must point to the shared library. The name argument is used to construct the name of the initialization function: an external C function called ‘`initname()`’ in the shared library is called. The optional file argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

**load_source**(name, pathname, file)

Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized again. The name argument is used to create or access a module object. The pathname argument points to the source file. The file argument is the source file, open for reading as text, from the beginning. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix ‘.pyc’ or ‘.pyo’) exists, it will be used instead of parsing the given source file.

### 3.21.1 Examples

The following function emulates what was the standard import statement up to Python 1.4 (i.e., no hierarchical module names). (This implementation wouldn’t work in that version, since `find_module()` has been extended and `load_module()` has been added in 1.4.)
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()

A more complete example that implements hierarchical module names and includes a reload() function can be found in the standard module knee (which is intended as an example only — don’t rely on any part of it being a standard interface).

3.22 code — Interpreter base classes

The code module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class InteractiveInterpreter([locals])
    This class deals with parsing and interpreter state (the user’s namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional locals argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key '__name__' set to '__console__' and key '__doc__' set to None.

class InteractiveConsole([locals[, filename]])
    Closely emulate the behavior of the interactive Python interpreter. This class builds on InteractiveInterpreter and adds prompting using the familiar sys.ps1 and sys.ps2, and input buffering.

interact([[banner[, readfunc[, local]]]])
    Convenience function to run a read-eval-print loop. This creates a new instance of InteractiveConsole and sets readfunc to be used as the raw_input() method, if provided. If local is provided, it is passed to the InteractiveConsole constructor for use as the default namespace for the interpreter loop. The interact() method of the instance is then run with banner passed as the banner to use, if provided. The console object is discarded after use.

compile_command([source[, filename[, symbol]]])
    This function is useful for programs that want to emulate Python’s interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function almost always makes the same decision as the real interpreter main loop.

source is the source string; filename is the optional filename from which source was read, defaulting to '<in-
put’; and symbol is the optional grammar start symbol, which should be either ‘single’ (the default) or ‘eval’.

Returns a code object (the same as compile(source, filename, symbol)) if the command is complete and valid; None if the command is incomplete; raises SyntaxError if the command is complete and contains a syntax error, or raises OverflowError if the command includes a numeric constant which exceeds the range of the appropriate numeric type.

### 3.22.1 Interactive Interpreter Objects

**runsource** *(source[, filename[, symbol]])*

Compile and run some source in the interpreter. Arguments are the same as for compile_command(); the default for filename is ‘<input>’, and for symbol is ‘single’. One several things can happen:

- The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method. runsource() returns 0.
- The input is incomplete, and more input is required; compile_command() returned None. runsource() returns 1.
- The input is complete; compile_command() returned a code object. The code is executed by calling the runcode() (which also handles run-time exceptions, except for SystemExit). runsource() returns 0.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

**runcode** *(code)*

Execute a code object. When an exception occurs, showtraceback() is called to display a traceback. All exceptions are caught except SystemExit, which is allowed to propagate.

A note about KeyboardInterrupt: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

**showsyntaxerror** *(filename)*

Display the syntax error that just occurred. This does not display a stack trace because there isn’t one for syntax errors. If filename is given, it is stuffed into the exception instead of the default filename provided by Python’s parser, because it always uses ‘<string>’ when reading from a string. The output is written by the write() method.

**showtraceback** *

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the write() method.

**write** *(data)*

Write a string to the standard error stream (sys.stderr). Derived classes should override this to provide the appropriate output handling as needed.

### 3.22.2 Interactive Console Objects

The InteractiveConsole class is a subclass of InteractiveInterpreter, and so offers all the methods of the interpreter objects as well as the following additions.

**interact** *(banner)*

Closely emulate the interactive Python console. The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it’s so close!).
push (line)
Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal
newlines. The line is appended to a buffer and the interpreter’s runsource() method is called with the
concatenated contents of the buffer as source. If this indicates that the command was executed or invalid,
the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was
appended. The return value is 1 if more input is required, 0 if the line was dealt with in some way (this is the
same as runsource()).
resetbuffer()
Remove any unhandled source text from the input buffer.
raw_input ([prompt])
Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the
EOF key sequence, EOFError is raised. The base implementation uses the built-in function raw_input();
a subclass may replace this with a different implementation.

3.23 codeop — Compile Python code

The codeop module provides a function to compile Python code with hints on whether it is certainly complete,
possibly complete or definitely incomplete. This is used by the code module and should not normally be used
directly.

The codeop module defines the following function:

```
codeop.compile_command (source[, filename[, symbol]])
```

Tries to compile source, which should be a string of Python code and return a code object if source is valid
Python code. In that case, the filename attribute of the code object will be filename, which defaults to '<input>'.
Returns None if source is not valid Python code, but is a prefix of valid Python code.

If there is a problem with source, an exception will be raised. SyntaxError is raised if there is invalid Python
syntax, and OverflowError if there is an invalid numeric constant.

The symbol argument determines whether source is compiled as a statement ('single', the default) or as an
expression ('eval'). Any other value will cause ValueError to be raised.

Caveat: It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching
the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a
backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for
the parser is better.

3.24 pprint — Data pretty printer

The pprint module provides a capability to “pretty-print” arbitrary Python data structures in a form which can be
used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types,
the representation may not be loadable. This may be the case if objects such as files, sockets, classes, or instances are
included, as well as many other builtin objects which are not representable as Python constants.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don’t
fit within the allowed width. Construct PrettyPrinter objects explicitly if you need to adjust the width constraint.

The pprint module defines one class:

```
class PrettyPrinter (...)
```

Construct a PrettyPrinter instance. This constructor understands several keyword parameters. An output
stream may be set using the stream keyword; the only method used on the stream object is the file protocol’s
write() method. If not specified, the PrettyPrinter adopts sys.stdout. Three additional parameters
may be used to control the formatted representation. The keywords are indent, depth, and width. The amount
of indentation added for each recursive level is specified by \texttt{indent}; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by \texttt{depth}; if the data structure being printed is too deep, the next contained level is replaced by ‘...’. By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the \texttt{width} parameter; the default is eighty characters. If a structure cannot be formatted within the constrained width, a best effort will be made.

```python
>>> import pprint, sys
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[ ['/usr/local/lib/python1.5',
  '/usr/local/lib/python1.5/test',
  '/usr/local/lib/python1.5/sunos5',
  '/usr/local/lib/python1.5/sharedmodules',
  '/usr/local/lib/python1.5/tkinter'],
  ''
]
>>> import parser
>>> tup = parser.ast2tuple(
  ... parser.suite(open('pprint.py').read()))[1][1][1]
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
(266, (267, (307, (287, (288, (...)))))
```

The \texttt{PrettyPrinter} class supports several derivative functions:

\textbf{pformat (object)}

Return the formatted representation of \texttt{object} as a string. The default parameters for formatting are used.

\textbf{pprint (object, stream )}

Prints the formatted representation of \texttt{object} on \texttt{stream}, followed by a newline. If \texttt{stream} is omitted, \texttt{sys.stdout} is used. This may be used in the interactive interpreter instead of a \texttt{print} statement for inspecting values. The default parameters for formatting are used.

```python
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=869440>,

```

\textbf{isreadable (object)}

Determine if the formatted representation of \texttt{object} is “readable,” or can be used to reconstruct the value using \texttt{eval()}. This always returns false for recursive objects.
>>> pprint.isreadable(stuff)
0

isrecursive (object)
Determine if object requires a recursive representation.

One more support function is also defined:

saferepr (object)
Return a string representation of object, protected against recursive data structures. If the representation of object exposes a recursive entry, the recursive reference will be represented as '<<Recursion on typename with id=number>>'. The representation is not otherwise formatted.

>>> pprint.saferepr(stuff)
"[<<Recursion on list with id=689268>, '', '/usr/local/lib/python1.5', '/usr/local/lib/python1.5/test', '/usr/local/lib/python1.5/sunos5', '/usr/local/lib/python1.5/sharedmodules', '/usr/local/lib/python1.5/tkinter']"

3.24.1 PrettyPrinter Objects

PrettyPrinter instances have the following methods:

pformat (object)
Return the formatted representation of object. This takes into Account the options passed to the PrettyPrinter constructor.

pprint (object)
Print the formatted representation of object on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new PrettyPrinter objects don’t need to be created.

isreadable (object)
Determine if the formatted representation of the object is “readable,” or can be used to reconstruct the value using eval(). Note that this returns false for recursive objects. If the depth parameter of the PrettyPrinter is set and the object is deeper than allowed, this returns false.

isrecursive (object)
Determine if the object requires a recursive representation.

3.25 repr — Alternate repr() implementation

The repr module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class Repr ()
Class which provides formatting services useful in implementing functions similar to the built-in repr(); size limits for different object types are added to avoid the generation of representations which are excessively long.

aRepr
This is an instance of Repr which is used to provide the repr() function described below. Changing the attributes of this object will affect the size limits used by repr() and the Python debugger.
This is the `repr()` method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

### 3.25.1 Repr Objects

`Repr` instances provide several members which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

- **maxlevel**
  - Depth limit on the creation of recursive representations. The default is 6.

- **maxdict**
  - **maxlist**
  - **maxtuple**
    - Limits on the number of entries represented for the named object type. The default for `maxdict` is 4, for the others, 6.

- **maxlong**
  - Maximum number of characters in the representation for a long integer. Digits are dropped from the middle. The default is 40.

- **maxstring**
  - Limit on the number of characters in the representation of the string. Note that the “normal” representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

- **maxother**
  - This limit is used to control the size of object types for which no specific formatting method is available on the `Repr` object. It is applied in a similar manner as `maxstring`. The default is 20.

- **repr**(obj)
  - The equivalent to the built-in `repr()` that uses the formatting imposed by the instance.

- **repr1**(obj, level)
  - Recursive implementation used by `repr()`. This uses the type of `obj` to determine which formatting method to call, passing it `obj` and `level`. The type-specific methods should call `repr1()` to perform recursive formatting, with `level - 1` for the value of `level` in the recursive call.

- **repr_type**(obj, level)
  - Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, `type` is replaced by `string.join(string.split(type(obj).__name__, '_'))`. Dispatch to these methods is handled by `repr1()`. Type-specific methods which need to recursively format a value should call `selfrepr1(subobj, level - 1)`.

### 3.25.2 Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:
import repr
import sys

class MyRepr(repr.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return 'obj'

aRepr = MyRepr()
print aRepr.repr(sys.stdin)  # prints '<stdin>'

3.26 new — Creation of runtime internal objects

The new module allows an interface to the interpreter object creation functions. This is for use primarily in marshal-type functions, when a new object needs to be created "magically" and not by using the regular creation functions. This module provides a low-level interface to the interpreter, so care must be exercised when using this module.

The new module defines the following functions:

instance (class, dict)
This function creates an instance of class with dictionary dict without calling the __init__() constructor. If dict is omitted or None, a new, empty dictionary is created for the new instance. Note that there are no guarantees that the object will be in a consistent state.

instancemethod (function, instance, class)
This function will return a method object, bound to instance, or unbound if instance is None. function must be callable, and instance must be an instance object or None.

function (code, globals, name, argdefs)
Returns a (Python) function with the given code and globals. If name is given, it must be a string or None. If it is a string, the function will have the given name, otherwise the function name will be taken from code.co_name. If argdefs is given, it must be a tuple and will be used to determine the default values of parameters.

code (argcount, nlocals, stacksize, flags, codestring, constants, names, varnames, filename, name, firstlineno, lnotab)
This function is an interface to the PyCode_New() C function.

module (name)
This function returns a new module object with name name. name must be a string.

classobj (name, baseclasses, dict)
This function returns a new class object, with name name, derived from baseclasses (which should be a tuple of classes) and with namespace dict.

3.27 site — Site-specific configuration hook

This module is automatically imported during initialization.

In earlier versions of Python (up to and including 1.5a3), scripts or modules that needed to use site-specific modules would place 'import site' somewhere near the top of their code. This is no longer necessary.

This will append site-specific paths to the module search path.
It starts by constructing up to four directories from a head and a tail part. For the head part, it uses `sys.prefix` and `sys.exec_prefix`; empty heads are skipped. For the tail part, it uses the empty string (on Macintosh or Windows) or it uses first ‘lib/python2.1/site-packages’ and then ‘lib/site-python’ (on UNIX). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds to `sys.path`, and also inspects the path for configuration files.

A path configuration file is a file whose name has the form ‘package.pth’; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, but no check is made that the item refers to a directory (rather than a file). No item is added to `sys.path` more than once. Blank lines and lines beginning with `#` are skipped. Lines starting with `import` are executed.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to ‘/usr/local’. The Python 2.1 library is then installed in ‘/usr/local/lib/python2.1’ (where only the first three characters of `sys.version` are used to form the installation path name). Suppose this has a subdirectory ‘/usr/local/lib/python2.1/site-packages’ with three subsubdirectories, ‘foo’, ‘bar’ and ‘spam’, and two path configuration files, ‘foo.pth’ and ‘bar.pth’. Assume ‘foo.pth’ contains the following:

```
# foo package configuration
foo
bar
bletch
```

and ‘bar.pth’ contains:

```
# bar package configuration
bar
```

Then the following directories are added to `sys.path`, in this order:

```
/usr/local/lib/python1.5/site-packages/bar
/usr/local/lib/python1.5/site-packages/foo
```

Note that ‘bletch’ is omitted because it doesn’t exist; the ‘bar’ directory precedes the ‘foo’ directory because ‘bar.pth’ comes alphabetically before ‘foo.pth’; and ‘spam’ is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. If this import fails with an `ImportError` exception, it is silently ignored.

Note that for some non-UNIX systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` is still attempted.

### 3.28 user — User-specific configuration hook

As a policy, Python doesn’t run user-specified code on startup of Python programs. (Only interactive sessions execute the script specified in the `PYTHONSTARTUP` environment variable if it exists).

However, some programs or sites may find it convenient to allow users to have a standard customization file, which gets run when a program requests it. This module implements such a mechanism. A program that wishes to use the mechanism must execute the statement
import user

The user module looks for a file `.pythonrc.py’ in the user’s home directory and if it can be opened, executes it (using `execfile()’ in its own (i.e. the module user’s) global namespace. Errors during this phase are not caught; that’s up to the program that imports the user module, if it wishes. The home directory is assumed to be named by the HOME environment variable; if this is not set, the current directory is used.

The user’s `.pythonrc.py’ could conceivably test for `sys.version’ if it wishes to do different things depending on the Python version.

A warning to users: be very conservative in what you place in your `.pythonrc.py’ file. Since you don’t know which programs will use it, changing the behavior of standard modules or functions is generally not a good idea.

A suggestion for programmers who wish to use this mechanism: a simple way to let users specify options for your package is to have them define variables in their `.pythonrc.py’ file that you test in your module. For example, a module spam that has a verbosity level can look for a variable `user.spam_verbose’, as follows:

```python
import user
try:
    verbose = user.spam_verbose  # user’s verbosity preference
except AttributeError:
    verbose = 0  # default verbosity
```

Programs with extensive customization needs are better off reading a program-specific customization file.

Programs with security or privacy concerns should not import this module; a user can easily break into a program by placing arbitrary code in the `.pythonrc.py’ file.

Modules for general use should not import this module; it may interfere with the operation of the importing program.

See Also:
Module site (section 3.27):
Site-wide customization mechanism.

3.29 __builtin__ — Built-in functions

This module provides direct access to all ‘built-in’ identifiers of Python; e.g. __builtin__.open is the full name for the built-in function open(). See section 2.3, “Built-in Functions.”

3.30 __main__ — Top-level script environment

This module represents the (otherwise anonymous) scope in which the interpreter’s main program executes — commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic “conditional script” stanza causes a script to run:

```python
if __name__ == "__main__":
    main()
```
String Services

The modules described in this chapter provide a wide range of string manipulation operations. Here’s an overview:

- **string** — Common string operations.
- **re** — Regular expression search and match operations with a Perl-style expression syntax.
- **struct** — Interpret strings as packed binary data.
- **difflib** — Helpers for computing differences between objects.
- **fpformat** — General floating point formatting functions.
- **StringIO** — Read and write strings as if they were files.
- **cStringIO** — Faster version of **StringIO**, but not subclassable.
- **codecs** — Encode and decode data and streams.
- **unicodedata** — Access the Unicode Database.

### 4.1 string — Common string operations

This module defines some constants useful for checking character classes and some useful string functions. See the module **re** for string functions based on regular expressions.

The constants defined in this module are:

- **digits**
  - The string ‘0123456789’.

- **hexdigits**
  - The string ‘0123456789abcdefABCDEF’.

- **letters**
  - The concatenation of the strings **lowercase** and **uppercase** described below.

- **lowercase**
  - A string containing all the characters that are considered lowercase letters. On most systems this is the string ‘abcdefgijklmnopqrstuvwxyz’. Do not change its definition — the effect on the routines **upper()** and **swapcase()** is undefined.

- **octdigits**
  - The string ‘01234567’.

- **punctuation**
  - String of ASCII characters which are considered punctuation characters in the ‘C’ locale.

- **printable**
  - String of characters which are considered printable. This is a combination of **digits**, **letters**, **punctuation**, and **whitespace**.

- **uppercase**
A string containing all the characters that are considered uppercase letters. On most systems this is the string ‘ABCDEFGHIJKLMNOPQRSTUVWXYZ’. Do not change its definition — the effect on the routines lower() and swapcase() is undefined.

**whitespace**

A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab. Do not change its definition — the effect on the routines strip() and split() is undefined.

Many of the functions provided by this module are also defined as methods of string and Unicode objects; see “String Methods” (section 2.1.5) for more information on those. The functions defined in this module are:

**atof** *(s)*

*Deprecated since release 2.0.* Use the float() built-in function.

Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign (‘+’ or ‘-’). Note that this behaves identical to the built-in function float() when passed a string.

**atoi** *(s*, *base)*

*Deprecated since release 2.0.* Use the int() built-in function.

Convert string s to an integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (‘+’ or ‘-’). The base defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): ‘0x’ or ‘0X’ means 16, ‘0’ means 8, anything else means 10. If base is 16, a leading ‘0x’ or ‘0X’ is always accepted, though not required. This behaves identically to the built-in function int() when passed a string. (Also note: for a more flexible interpretation of numeric literals, use the built-in function eval().)

**atol** *(s*, *base)*

*Deprecated since release 2.0.* Use the long() built-in function.

Convert string s to a long integer in the given base. The string must consist of one or more digits, optionally preceded by a sign (‘+’ or ‘-’). The base argument has the same meaning as for atoi(). A trailing ‘l’ or ‘L’ is not allowed, except if the base is 0. Note that when invoked without base or with base set to 10, this behaves identically to the built-in function long() when passed a string.

**capitalize** *(word)*

Capitalize the first character of the argument.

**capwords** *(s)*

Split the argument into words using split(), capitalize each word using capitalize(), and join the capitalized words using join(). Note that this replaces runs of whitespace characters by a single space, and removes leading and trailing whitespace.

**expandtabs** *(s*, *tabsize)*

Expand tabs in a string, i.e. replace them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn’t understand other non-printing characters or escape sequences. The tab size defaults to 8.

**find** *(s, sub*, *start*, *end)*

Return the lowest index in s where the substring sub is found such that sub is wholly contained in s[start:end]. Return -1 on failure. Defaults for start and end and interpretation of negative values is the same as for slices.

**rfind** *(s, sub*, *start*, *end)*

Like find() but find the highest index.

**index** *(s, sub*, *start*, *end)*

Like find() but raise ValueError when the substring is not found.
**rindex** *(s, sub[, start[, end]])*

Like `rfind()` but raise `ValueError` when the substring is not found.

**count** *(s, sub[, start[, end]])*

Return the number of (non-overlapping) occurrences of substring `sub` in string `s[start:end]`. Defaults for `start` and `end` and interpretation of negative values are the same as for slices.

**lower** *(s)*

Return a copy of `s`, but with upper case letters converted to lower case.

**maketrans** *(from, to)*

Return a translation table suitable for passing to `translate()` or `regex.compile()`, that will map each character in `from` into the character at the same position in `to`; `from` and `to` must have the same length.

**Warning:** don’t use strings derived from `lowercase` and `uppercase` as arguments; in some locales, these don’t have the same length. For case conversions, always use `lower()` and `upper()`.

**split** *(s[, sep[, maxsplit]])*

Return a list of the words of the string `s`. If the optional second argument `sep` is absent or `None`, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed). If the second argument `sep` is present and not `None`, it specifies a string to be used as the word separator. The returned list will then have one more item than the number of non-overlapping occurrences of the separator in the string. The optional third argument `maxsplit` defaults to 0. If it is nonzero, at most `maxsplit` number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most `maxsplit+1` elements).

**splitfields** *(s[, sep[, maxsplit]])*

This function behaves identically to `split()`. (In the past, `split()` was only used with one argument, while `splitfields()` was only used with two arguments.)

**join** *(words[, sep])* 

Concatenate a list or tuple of words with intervening occurrences of `sep`. The default value for `sep` is a single space character. It is always true that `string.join(string.split(s, sep), sep)` equals `s`.

**joinfields** *(words[, sep])* 

This function behaves identical to `join()`. (In the past, `join()` was only used with one argument, while `joinfields()` was only used with two arguments.)

**lstrip** *(s)*

Return a copy of `s` but without leading whitespace characters.

**rstrip** *(s)*

Return a copy of `s` but without trailing whitespace characters.

**strip** *(s)*

Return a copy of `s` without leading or trailing whitespace.

**swapcase** *(s)*

Return a copy of `s`, but with lower case letters converted to upper case and vice versa.

**translate** *(s, table[, deletechars])* 

Delete all characters from `s` that are in `deletechars` (if present), and then translate the characters using `table`, which must be a 256-character string giving the translation for each character value, indexed by its ordinal.

**upper** *(s)*

Return a copy of `s`, but with lower case letters converted to upper case.

**ljust** *(s, width)*

**rjust** *(s, width)*

**center** *(s, width)*

These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least `width` characters wide, created by padding the string `s` with spaces until the given width on
the right, left or both sides. The string is never truncated.

```
zfill(s, width)
```

Pad a numeric string on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly.

```
replace(str, old, new[, maxsplit])
```

Return a copy of string `str` with all occurrences of substring `old` replaced by `new`. If the optional argument `maxsplit` is given, the first `maxsplit` occurrences are replaced.

This module is implemented in Python. Much of its functionality has been reimplemented in the built-in module `strop`. However, you should never import the latter module directly. When `string` discovers that `strop` exists, it transparently replaces parts of itself with the implementation from `strop`. After initialization, there is no overhead in using `string` instead of `strop`.

### 4.2 re — Regular expression operations

This module provides regular expression matching operations similar to those found in Perl. Regular expression pattern strings may not contain null bytes, but can specify the null byte using the `\number` notation. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings. The `re` module is always available.

Regular expressions use the backslash character (`\`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `\` as the pattern string, because the regular expression must be `\`, and each backslash must be expressed as `\` inside a regular Python string literal.

The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `r`. So `r"\n"` is a two-character string containing `\` and `n`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

**Implementation note:** The `re` module has two distinct implementations: `sre` is the default implementation and includes Unicode support, but may run into stack limitations for some patterns. Though this will be fixed for a future release of Python, the older implementation (without Unicode support) is still available as the `pre` module.

**See Also:**

- *Mastering Regular Expressions*
  
  Book on regular expressions by Jeffrey Friedl, published by O’Reilly. The Python material in this book dates from before the `re` module, but it covers writing good regular expression patterns in great detail.

#### 4.2.1 Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if `A` and `B` are both regular expressions, then `AB` is also an regular expression. If a string `p` matches `A` and another string `q` matches `B`, the string `pq` will match `AB`. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced below, or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the Regular Expression HOWTO, accessible from [http://www.python.org/doc/howto/](http://www.python.org/doc/howto/).
Regular expressions can contain both special and ordinary characters. Most ordinary characters, like ‘A’, ‘a’, or ‘0’, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so \[last\] matches the string ‘last’.

In the rest of this section, we’ll write RE’s in this special style, usually without quotes, and strings to be matched ‘in single quotes’.

Some characters, like ‘|’ or ‘(’, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

The special characters are:

‘.’ (Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

‘*’ (Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

‘$’ Matches the end of the string, and in MULTILINE mode also matches before a newline. \[foo\$\] matches both ‘foo’ and ‘foobar’, while the regular expression ‘foo$’ matches only ‘foo’.

‘*’ Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. \[ab*\] will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

‘+’ Causes the resulting RE to match 1 or more repetitions of the preceding RE. \[ab+\] will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

‘?’ Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. \[ab?\] will match either ‘a’ or ‘ab’.

The ‘*’, ‘+’, and ‘?’ qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn’t desired; if the RE ‘<.*>’ is matched against ‘<H1>\title</H1>’

, it will match the entire string, and not just ‘<H1>’.

Adding ‘?’ after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using ‘[.*]’ in the previous expression will match only ‘<H1>’.

\(m, n\) Causes the resulting RE to match from \(m\) to \(n\) repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, \[a\{3, 5\}\] will match from 3 to 5 ‘a’ characters. Omitting \(n\) specifies an infinite upper bound; you can’t omit \(m\).

\(m, n\)? Causes the resulting RE to match from \(m\) to \(n\) repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string ‘aaaaaa’, \[a\{3, 5\}\] will match 5 ‘a’ characters, while \[a\{3, 5\}\?\] will only match 3 characters.

‘\’ Either escapes special characters (permitting you to match characters like ‘*’, ‘?’, and so forth), or signals a special sequence; special sequences are discussed below.

If you’re not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn’t recognized by Python’s parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it’s highly recommended that you use raw strings for all but the simplest expressions.

[] Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a ‘–’. Special characters are not active inside sets. For example, \[akm$\]\ will match any of the characters ‘a’, ‘k’, ‘m’, or ‘$’; \[a-zA-Z0-9\] matches any letter or digit. Character classes such as ‘\w’ or ‘\s’ (defined below) are also acceptable inside a range. If you want to include a ‘]’ or a ‘–’ inside a set, precede it with a backslash, or place it as the first character. The pattern ‘[[ ]]’ will match ‘]’, for example.
You can match the characters not within a range by complementing the set. This is indicated by including a `^` as the first character of the set; `^` elsewhere will simply match the `^` character. For example, `^[\^5]` will match any character except `^5`.

`|` A | B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the `|` in this way. This can be used inside groups (see below) as well. REs separated by `|` are tried from left to right, and the first one that allows the complete pattern to match is considered the accepted branch. This means that if A matches, B will never be tested, even if it would produce a longer overall match. In other words, the `|` operator is never greedy. To match a literal `|`, use `\|`, or enclose it inside a character class, as in `^[ ]`.  

(... Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\{number\}` special sequence, described below. To match the literals `( ` or `)` , use `\( ` or `\)` , or enclose them inside a character class: `^[ ]`.  

(?? Matches whatever text was matched by the earlier group named `id` in the example above can also be referenced as the numbered group 1.  

For example, if the pattern is `^[?P<id>[a-zA-Z_\-]+\w*]`, the group can be referenced by its name in arguments to methods of match objects, such as `m.group(\'id\')` or `m.end(\'id\')`, and also by name in pattern text (e.g. `^[?P=id]*`) and replacement text (e.g. `\g<id>`).  

(??= Matches whatever text was matched by the earlier group named `name`.  

(??= Matches if `[...]` matches next, but doesn’t consume any of the string. This is called a lookahead assertion. For example, `Isaac (?=Asimov)` will match `Isaac` only if it’s followed by `Asimov`.

(??! Matches if `[...]` doesn’t match next. This is a negative lookahead assertion. For example, `Isaac (?)Asimov)` will match `Isaac` only if it’s not followed by `Asimov`.

(??<= Matches if the current position in the string is preceded by a match for `[...]` that ends at the current position. This is called a positive lookbehind assertion. `^[?a<abc]def]` will match `abcdef`, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that `^abc|\a\b` are allowed, but `^a*` isn’t.
(?<!... Matches if the current position in the string is not preceded by a match for \... This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length.

The special sequences consist of `\` and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, `\$` matches the character `$`.

\number Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(\.+ \1)` matches `the the` or `55 55`, but not `the end` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of number is 0, or number is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value number. Inside the `[` and `]` of a character class, all numeric escapes are treated as characters.

\A Matches only at the start of the string.
\b Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character. Inside a character range, `\b` represents the backspace character, for compatibility with Python’s string literals.
\B Matches the empty string, but only when it is not at the beginning or end of a word.
\d Matches any decimal digit; this is equivalent to the set `[0-9]`.
\D Matches any non-digit character; this is equivalent to the set `[^0-9]`.
\s Matches any whitespace character; this is equivalent to the set `[\t\n\r\f\v]`.
\S Matches any non-whitespace character; this is equivalent to the set `[^ \t\n\r\f\v]`.
\w When the LOCALE and UNICODE flags are not specified, matches any alphanumeric character; this is equivalent to the set `[a-zA-Z0-9_]`. With LOCALE, it will match the set `[0-9_]` plus whatever characters are defined as letters for the current locale. If UNICODE is set, this will match the characters `[0-9_]` plus whatever is classified as alphanumeric in the Unicode character properties database.
\W When the LOCALE and UNICODE flags are not specified, matches any non-alphanumeric character; this is equivalent to the set `[^a-zA-Z0-9_]`. With LOCALE, it will match any character not in the set `[0-9_]`, and not defined as a letter for the current locale. If UNICODE is set, this will match anything other than `[0-9_]` and characters marked at alphanumeric in the Unicode character properties database.
\Z Matches only at the end of the string.
\\ Matches a literal backslash.

4.2.2 Matching vs. Searching

Python offers two different primitive operations based on regular expressions: match and search. If you are accustomed to Perl’s semantics, the search operation is what you’re looking for. See the `search()` function and corresponding method of compiled regular expression objects.

Note that match may differ from search using a regular expression beginning with `\^`: `\^` matches only at the start of the string, or in MULTILINE mode also immediately following a newline. The “match” operation succeeds only if the pattern matches at the start of the string regardless of mode, or at the starting position given by the optional pos argument regardless of whether a newline precedes it.

4.2. re — Regular expression operations 83
4.2.3 Module Contents

The module defines the following functions and constants, and an exception:

```
re.compile("a").match("ba", 1)  # succeeds
re.compile("a").search("ba", 1)  # fails; ‘a’ not at start
re.compile("a").search("\na", 1)  # fails; ‘a’ not at start
re.compile("a", re.M).search("\na", 1)  # succeeds
re.compile("a", re.M).search("ba", 1)  # fails; no preceding \n
```

```
4.2.3 Module Contents

The module defines the following functions and constants, and an exception:

```
        compile (pattern[, flags])

        Compile a regular expression pattern into a regular expression object, which can be used for matching using its
        match() and search() methods, described below.

        The expression’s behaviour can be modified by specifying a flags value. Values can be any of the following
        variables, combined using bitwise OR (the | operator).

        The sequence

            prog = re.compile(pat)
            result = prog.match(str)

        is equivalent to

            result = re.match(pat, str)

        but the version using compile() is more efficient when the expression will be used several times in a single
        program.

        I
        IGNORECASE
        Perform case-insensitive matching; expressions like '^[A-Z]' will match lowercase letters, too. This is not
        affected by the current locale.

        L
        LOCALE
        Make '\w', '\W', '\b', and '\B' dependent on the current locale.

        M
        MULTILINE
        When specified, the pattern character ‘^’ matches at the beginning of the string and at the beginning of each
        line (immediately following each newline); and the pattern character ‘$’ matches at the end of the string and at
        the end of each line (immediately preceding each newline). By default, ‘^’ matches only at the beginning of the
        string, and ‘$’ only at the end of the string and immediately before the newline (if any) at the end of the string.

        S
        DOTALL
        Make the ‘.’ special character match any character at all, including a newline; without this flag, ‘.’ will match
        anything except a newline.

        U
        UNICODE

        X
```

84 Chapter 4. String Services
VERBOSE
This flag allows you to write regular expressions that look nicer. Whitespace within the pattern is ignored, except when in a character class or preceded by an unescaped backslash, and, when a line contains a '#' neither in a character class or preceded by an unescaped backslash, all characters from the leftmost such '#' through the end of the line are ignored.

search \( \text{pattern, string} [flags] \)
Scan through \textit{string} looking for a location where the regular expression \textit{pattern} produces a match, and return a corresponding \textit{MatchObject} instance. Return \texttt{None} if no position in the \textit{string} matches the \textit{pattern}; note that this is different from finding a zero-length match at some point in the \textit{string}.

match \( \text{pattern, string} [flags] \)
If zero or more characters at the beginning of \textit{string} match the regular expression \textit{pattern}, return a corresponding \textit{MatchObject} instance. Return \texttt{None} if the \textit{string} does not match the \textit{pattern}; note that this is different from a zero-length match.

Note: If you want to locate a match anywhere in \textit{string}, use \texttt{search()} instead.

split \( \text{pattern, string} [maxsplit = 0] \)
Split \textit{string} by the occurrences of \textit{pattern}. If capturing parentheses are used in \textit{pattern}, then the text of all groups in the \textit{pattern} are also returned as part of the resulting list. If \textit{maxsplit} is nonzero, at most \textit{maxsplit} splits occur, and the remainder of the \textit{string} is returned as the final element of the list. (Incompatibility note: in the original Python 1.5 release, \textit{maxsplit} was ignored. This has been fixed in later releases.)

```python
>>> re.split('\\W+', 'Words, words, words."
[''Words'', 'words', 'words', '"

>>> re.split('\\W+', 'Words, words, words."
[''Words'', '', 'words', '', 'words', '"

>>> re.split('\\W+', 'Words, words, words.', 1)
[''Words'', 'words, words.']
```

This function combines and extends the functionality of the old \texttt{regsub.split()} and \texttt{regsub.splitx()}.

findall \( \text{pattern, string} \)
Return a list of all non-overlapping matches of \textit{pattern} in \textit{string}. If one or more groups are present in the \textit{pattern}, return a list of groups; this will be a list of tuples if the \textit{pattern} has more than one group. Empty matches are included in the result. New in version 1.5.2.

sub \( \text{pattern, repl, string} [count = 0] \)
Return the string obtained by replacing the leftmost non-overlapping occurrences of \textit{pattern} in \textit{string} by the replacement \textit{repl}. If the \textit{pattern} isn’t found, \textit{string} is returned unchanged. \textit{repl} can be a string or a function; if a function, it is called for every non-overlapping occurrence of \textit{pattern}. The function takes a single match object argument, and returns the replacement string. For example:

```python
>>> def dashrepl(matchobj):
.... if matchobj.group(0) == '-' : return ''
.... else: return '-'

>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
```

The pattern may be a string or a regex object; if you need to specify regular expression flags, you must use a regex object, or use embedded modifiers in a pattern; e.g. \\texttt{'sub("(?i)b+", "x", "bbbbb BBBB")'} returns ''xx''.

The optional argument \textit{count} is the maximum number of pattern occurrences to be replaced; \textit{count} must be a non-negative integer, and the default value of 0 means to replace all occurrences.

Empty matches for the pattern are replaced only when not adjacent to a previous match, so \\texttt{'sub(''x*'', '', '-', ' abc'')} returns ''-a-b-c-''.
If `repl` is a string, any backslash escapes in it are processed. That is, ‘\n’ is converted to a single newline character, ‘\r’ is converted to a linefeed, and so forth. Unknown escapes such as ‘\j’ are left alone. Backreferences, such as ‘\6’, are replaced with the substring matched by group 6 in the pattern.

In addition to character escapes and backreferences as described above, ‘\g<name>’ will use the substring matched by the group named ‘name’, as defined by the ‘{(?P<name>...)’ syntax. ‘\g<number>’ uses the corresponding group number; ‘\g<2>’ is therefore equivalent to ‘\2’, but isn’t ambiguous in a replacement such as ‘\g<2>0’. ‘\20’ would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character ‘0’.

```python
subn(pattern, repl, string[, count = 0])
Perform the same operation as sub(), but return a tuple (new_string, number_of_subs_made).
```

```python
escape(string)
Return string with all non-alphanumerics backslashed; this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.
```

```python
exception error
Exception raised when a string passed to one of the functions here is not a valid regular expression (e.g., unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern.
```

### 4.2.4 Regular Expression Objects

Compiled regular expression objects support the following methods and attributes:

```python
search(string[, pos[, endpos ]])
Scan through string looking for a location where this regular expression produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional pos and endpos parameters have the same meaning as for the match() method.
```

```python
match(string[, pos[, endpos ]])
If zero or more characters at the beginning of string match this regular expression, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Note: If you want to locate a match anywhere in string, use search() instead.

The optional second parameter pos gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the ‘ˆ’ pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter endpos limits how far the string will be searched; it will be as if the string is endpos characters long, so only the characters from pos to endpos will be searched for a match.
```

```python
split(string[, maxsplit = 0 ])
Identical to the split() function, using the compiled pattern.
```

```python
findall(string)
Identical to the findall() function, using the compiled pattern.
```

```python
sub(repl, string[, count = 0 ])
Identical to the sub() function, using the compiled pattern.
```

```python
subn(repl, string[, count = 0 ])
Identical to the subn() function, using the compiled pattern.
```

```python
flags
The flags argument used when the regex object was compiled, or 0 if no flags were provided.
```
A dictionary mapping any symbolic group names defined by `(?P<name>)` to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

The pattern string from which the regex object was compiled.

### 4.2.5 Match Objects

MatchObject instances support the following methods and attributes:

- **expand** *(template)*
  
  Return the string obtained by doing backslash substitution on the template string *template*, as done by the `sub()` method. Escapes such as \n’ are converted to the appropriate characters, and numeric backreferences (\1’, ‘\2’) and named backreferences (‘\g<1>’, ‘\g<name>’) are replaced by the contents of the corresponding group.

- **group** *(group1, ...)*
  
  Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, `group1` defaults to zero (i.e. the whole match is returned). If a `groupN` argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is -1. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

  If the regular expression uses the `(?P<name>)` syntax, the `groupN` arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an IndexError exception is raised.

  A moderately complicated example:

  ```
  m = re.match(r"(?P<int>\d+)\.(\d*)", '3.14')
  ```

  After performing this match, `m.group(1)` is ‘3’, as is `m.group('int')`, and `m.group(2)` is ‘14’.

- **groups** *(default)*
  
  Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None. (Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

- **groupdict** *(default)*
  
  Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None.

- **start** *(group)*
  
  Return the indices of the start and end of the substring matched by `group`; `group` defaults to zero (meaning the whole matched substring). Return -1 if `group` exists but did not contribute to the match. For a match object `m`, and a group `g` that did contribute to the match, the substring matched by group `g` (equivalent to `m.group(g)`) is

  ```
  m.string[m.start(g):m.end(g)]
  ```

  Note that `m.start(group)` will equal `m.end(group)` if `group` matched a null string. For example, after `m =`
re.search('b(c?)', 'cba'), m.start(0) is 1, m.end(0) is 2, m.start(1) and m.end(1) are both 2, and m.start(2) raises an IndexError exception.

span([group])
For MatchObject m, return the 2-tuple (m.start(group), m.end(group)). Note that if group did not contribute to the match, this is (-1, -1). Again, group defaults to zero.

pos
The value of pos which was passed to the search() or match() function. This is the index into the string at which the regex engine started looking for a match.

endpos
The value of endpos which was passed to the search() or match() function. This is the index into the string beyond which the regex engine will not go.

lastgroup
The name of the last matched capturing group, or None if the group didn’t have a name, or if no group was matched at all.

lastindex
The integer index of the last matched capturing group, or None if no group was matched at all.

re
The regular expression object whose match() or search() method produced this MatchObject instance.

string
The string passed to match() or search().

4.3 struct — Interpret strings as packed binary data

This module performs conversions between Python values and C structs represented as Python strings. It uses format strings (explained below) as compact descriptions of the lay-out of the C structs and the intended conversion to/from Python values. This can be used in handling binary data stored in files or from network connections, among other sources.

The module defines the following exception and functions:

exception error
Exception raised on various occasions; argument is a string describing what is wrong.

pack (fmt, v1, v2, ...)
Return a string containing the values v1, v2, ... packed according to the given format. The arguments must match the values required by the format exactly.

unpack (fmt, string)
Unpack the string (presumably packed by pack (fmt, ...)) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (i.e. len (string) must equal calcsize (fmt)).

calcsize (fmt)
Return the size of the struct (and hence of the string) corresponding to the given format.

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types:
Notes:

(1) The ‘I’ conversion code will convert to a Python long if the C int is the same size as a C long, which is typical on most modern systems. If a C int is smaller than a C long, an Python integer will be created instead.

A format character may be preceded by an integral repeat count; e.g. the format string ’4h’ means exactly the same as ’hhhh’.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the ‘s’ format character, the count is interpreted as the size of the string, not a repeat count like for the other format characters; e.g. ’10s’ means a single 10-byte string, while ’10c’ means 10 characters. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting string always has exactly the specified number of bytes. As a special case, ’0s’ means a single, empty string (while ’0c’ means 0 characters).

The ‘p’ format character can be used to encode a Pascal string. The first byte is the length of the stored string, with the bytes of the string following. If count is given, it is used as the total number of bytes used, including the length byte. If the string passed in to pack() is too long, the stored representation is truncated. If the string is too short, padding is used to ensure that exactly enough bytes are used to satisfy the count.

For the ‘I’ and ‘L’ format characters, the return value is a Python long integer.

For the ‘P’ format character, the return value is a Python integer or long integer, depending on the size needed to hold a pointer when it has been cast to an integer type. A NULL pointer will always be returned as the Python integer 0. When packing pointer-sized values, Python integer or long integer objects may be used. For example, the Alpha and Merced processors use 64-bit pointer values, meaning a Python long integer will be used to hold the pointer; other platforms use 32-bit pointers and will use a Python integer.

By default, C numbers are represented in the machine’s native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Byte order</th>
<th>Size and alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘@’</td>
<td>native</td>
<td>native</td>
</tr>
<tr>
<td>‘=’</td>
<td>native</td>
<td>standard</td>
</tr>
<tr>
<td>‘&lt;’</td>
<td>little-endian</td>
<td>standard</td>
</tr>
<tr>
<td>‘&gt;’</td>
<td>big-endian</td>
<td>standard</td>
</tr>
<tr>
<td>‘!’</td>
<td>network (= big-endian)</td>
<td>standard</td>
</tr>
</tbody>
</table>
If the first character is not one of these, ‘@’ is assumed.

Native byte order is big-endian or little-endian, depending on the host system (e.g. Motorola and Sun are big-endian; Intel and DEC are little-endian).

Native size and alignment are determined using the C compiler’s `sizeof` expression. This is always combined with native byte order.

Standard size and alignment are as follows: no alignment is required for any type (so you have to use pad bytes); `short` is 2 bytes; `int` and `long` are 4 bytes. `float` and `double` are 32-bit and 64-bit IEEE floating point numbers, respectively.

Note the difference between ‘@’ and ‘=’: both use native byte order, but the size and alignment of the latter is standardized.

The form ‘!’ is available for those poor souls who claim they can’t remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (i.e. force byte-swapping); use the appropriate choice of ‘<’ or ‘>’.

The ‘P’ format character is only available for the native byte ordering (selected as the default or with the ‘@’ byte order character). The byte order character ‘=’ chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the ‘P’ format is not available.

Examples (all using native byte order, size and alignment, on a big-endian machine):

```python
>>> from struct import *
>>> pack('hhll', 1, 2, 3, 4)
'b\x00\x01\x02\x03\x04'
>>> unpack('hhll', b'\x00\x01\x02\x03\x04')
(1, 2, 3, 4)
>>> calcsize('hhll')
8
```

Hint: to align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero, e.g. the format ‘11h01’ specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries. This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

See Also:

Module `array` (section 5.8):
   Packed binary storage of homogeneous data.
Module `xdrlib` (section 12.9):
   Packing and unpacking of XDR data.

### 4.4 difflib — Helpers for computing deltas

New in version 2.1.

`get_close_matches` *(word, possibilities[, n[, cutoff ]]*)

Return a list of the best “good enough” matches. `word` is a sequence for which close matches are desired (typically a string), and `possibilities` is a list of sequences against which to match `word` (typically a list of strings).

Optional argument `n` (default 3) is the maximum number of close matches to return; `n` must be greater than 0.

Optional argument `cutoff` (default 0.6) is a float in the range [0, 1]. Possibilities that don’t score at least that
similar to word are ignored.
The best (no more than n) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```python
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
```

```python
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
```

```python
>>> get_close_matches('apple', keyword.kwlist)
[]
```

```python
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

class SequenceMatcher(...)
This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name “gestalt pattern matching.” The idea is to find the longest contiguous matching subsequence that contains no “junk” elements (the Ratcliff and Obershelp algorithm doesn’t address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that “look right” to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. SequenceMatcher is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

See Also:

Pattern Matching: The Gestalt Approach
Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.

4.4.1 SequenceMatcher Objects
class SequenceMatcher([isjunk, a[, b]])
Optional argument isjunk must be None (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is “junk” and should be ignored. None is equivalent to passing

```python
lambda x: 0, i.e. no elements are ignored. For example, pass
lambda x: x in " \t"
```

if you’re comparing lines as sequences of characters, and don’t want to synch up on blanks or hard tabs.
The optional arguments a and b are sequences to be compared; both default to empty strings. The elements of both sequences must be hashable.

SequenceMatcher objects have the following methods:

set_seqs (a, b)
Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use set_seq2() to set the commonly used sequence once and call set_seq1() repeatedly, once for each of the other sequences.

set_seq1 (a)
Set the first sequence to be compared. The second sequence to be compared is not changed.
set_seq2 (b)
Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match (alo, ahi, blo, bhi)
Find longest matching block in a [alo:ahi] and b [blo:bhi].

If isjunk was omitted or None, get_longest_match () returns (i, j, k) such that a [i:i+k] is equal to b [j:j+k], where alo <= i <= i+k <= ahi and blo <= j <= j+k <= bhi. For all (i', j', k') meeting those conditions, the additional conditions k >= k', i <= i', and if i == i', j <= j' are also met. In other words, of all maximal matching blocks, return one that starts earliest in a, and of all those maximal matching blocks that start earliest in a, return the one that starts earliest in b.

```python
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
(0, 4, 5)
```

If isjunk was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here’s the same example as before, but considering blanks to be junk. That prevents ‘ abcd’ from matching the ‘ abcd’ at the tail end of the second sequence directly. Instead only the ‘ abcd’ can match, and matches the leftmost ‘ abcd’ in the second sequence:

```python
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
(1, 0, 4)
```

If no blocks match, this returns (alo, blo, 0).

get_matching_blocks ()
Return list of triples describing matching subsequences. Each triple is of the form (i, j, n), and means that a [i:i+n] == b [j:j+n]. The triples are monotonically increasing in i and j.

The last triple is a dummy, and has the value (len (a), len (b), 0). It is the only triple with n == 0.

```python
>>> s = SequenceMatcher (None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[(0, 0, 2), (3, 2, 2), (5, 4, 0)]
```

get_opcodes ()
Return list of 5-tuples describing how to turn a into b. Each tuple is of the form (tag, il, i2, j1, j2). The first tuple has il == j1 == 0, and remaining tuples have il equal to the i2 from the preceeding tuple, and, likewise, j1 equal to the previous j2.

The tag values are strings, with these meanings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'replace'</td>
<td>a [il:i2] should be replaced by b [j1:j2].</td>
</tr>
<tr>
<td>'delete'</td>
<td>a [il:i2] should be deleted. Note that j1 == j2 in this case.</td>
</tr>
<tr>
<td>'insert'</td>
<td>b [j1:j2] should be inserted at a [il:i1]. Note that il == i2 in this case.</td>
</tr>
<tr>
<td>'equal'</td>
<td>a [il:i2] == b [j1:j2] (the sub-sequences are equal).</td>
</tr>
</tbody>
</table>

For example:
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
... print("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
... (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
  delete a[0:1] (q) b[0:0] ()
  equal a[1:3] (ab) b[0:2] (ab)
  replace a[3:4] (x) b[2:3] (y)

c::ratio()
nReturn a measure of the sequences' similarity as a float in the range [0, 1].

   Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T.
   Note that this is 1. if the sequences are identical, and 0. if they have nothing in common.

   This is expensive to compute if get_matching_blocks() or get_opcodes() hasn't already been called, in which case you may want to try quick_ratio() or real_quick_ratio() first to get an upper bound.

c::quick_ratio()
nReturn an upper bound on ratio() relatively quickly.

   This isn't defined beyond that it is an upper bound on ratio(), and is faster to compute.

c::real_quick_ratio()
nReturn an upper bound on ratio() very quickly.

   This isn't defined beyond that it is an upper bound on ratio(), and is faster to compute than either ratio() or quick_ratio().

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although quick_ratio() and real_quick_ratio() are always at least as large as ratio():

>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0

4.4.2 Examples

This example compares two strings, considering blanks to be "junk:"

>>> s = SequenceMatcher(lambda x: x == " ",
...     "private Thread currentThread;",
...     "private volatile Thread currentThread;")

ratio() returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a ratio() value over 0.6 means the sequences are close matches:
If you're only interested in where the sequences match, `get_matching_blocks()` is handy:

```python
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 6 elements
a[14] and b[23] match for 15 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use `get_opcodes()`:

```python
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
equal a[8:14] b[17:23]
equal a[14:29] b[23:38]
```

See `Tools/scripts/ndiff.py` from the Python source distribution for a fancy human-friendly file differencer, which uses `SequenceMatcher` both to view files as sequences of lines, and lines as sequences of characters.

See also the function `get_close_matches()` in this module, which shows how simple code building on `SequenceMatcher` can be used to do useful work.

### 4.5 fpformat — Floating point conversions

The `fpformat` module defines functions for dealing with floating point numbers representations in 100% pure Python. **Note:** This module is unneeded: everything here could be done via the `%` string interpolation operator.

The `fpformat` module defines the following functions and an exception:

#### fix(x, digs)
Format `x` as `[-]ddd.ddd` with `digs` digits after the point and at least one digit before. If `digs <= 0`, the decimal point is suppressed.

- `x` can be either a number or a string that looks like one. `digs` is an integer.
- Return value is a string.

#### sci(x, digs)
Format `x` as `[ -]d.dddE[+-]ddd` with `digs` digits after the point and exactly one digit before. If `digs <= 0`, one digit is kept and the point is suppressed.

- `x` can be either a real number, or a string that looks like one. `digs` is an integer.
- Return value is a string.

**exception NotANumber**
Exception raised when a string passed to `fix()` or `sci()` as the `x` parameter does not look like a number. This
is a subclass of `ValueError` when the standard exceptions are strings. The exception value is the improperly formatted string that caused the exception to be raised.

Example:

```python
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```

## 4.6 `StringIO` — Read and write strings as files

This module implements a file-like class, `StringIO`, that reads and writes a string buffer (also known as *memory files*). See the description of file objects for operations (section 2.1.7).

```python
class StringIO([buffer])
```

When a `StringIO` object is created, it can be initialized to an existing string by passing the string to the constructor. If no string is given, the `StringIO` will start empty.

The `StringIO` object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (i.e., that use the 8th bit) will cause a `UnicodeError` to be raised when `getvalue()` is called.

The following methods of `StringIO` objects require special mention:

### `getvalue()`

Retrieve the entire contents of the “file” at any time before the `StringIO` object’s `close()` method is called. See the note above for information about mixing Unicode and 8-bit strings; such mixing can cause this method to raise `UnicodeError`.

### `close()`

Free the memory buffer.

## 4.7 `cStringIO` — Faster version of `StringIO`

The module `cStringIO` provides an interface similar to that of the `StringIO` module. Heavy use of `StringIO.StringIO` objects can be made more efficient by using the function `StringIO()` from this module instead.

Since this module provides a factory function which returns objects of built-in types, there’s no way to build your own version using subclassing. Use the original `StringIO` module in that case.

Unlike the memory files implemented by the `StringIO` module, those provided by this module are not able to accept Unicode strings that cannot be encoded as plain ASCII strings.

The following data objects are provided as well:

#### `InputType`

The type object of the objects created by calling `StringIO` with a string parameter.

#### `OutputType`

The type object of the objects returned by calling `StringIO` with no parameters.

There is a C API to the module as well; refer to the module source for more information.
4.8 **codecs — Codec registry and base classes**

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry which manages the codec lookup process.

It defines the following functions:

**register (search_function)**

Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a tuple of functions \((encoder, decoder, stream_reader, stream_writer)\) taking the following arguments:

- **encoder** and **decoder**: These must be functions or methods which have the same interface as the \(\text{encode}()\)/\(\text{decode}()\) methods of Codec instances (see Codec Interface). The functions/methods are expected to work in a stateless mode.

- **stream_reader** and **stream_writer**: These have to be factory functions providing the following interface:
  
  \[
  \text{factory}(\text{stream}, \text{errors}='\text{strict}')
  \]

  The factory functions must return objects providing the interfaces defined by the base classes \(\text{StreamWriter}\) and \(\text{StreamReader}\), respectively. Stream codecs can maintain state.

  Possible values for errors are \(\text{`strict'}\) (raise an exception in case of an encoding error), \(\text{`replace'}\) (replace malformed data with a suitable replacement marker, such as ‘?’) and \(\text{‘ignore’}\) (ignore malformed data and continue without further notice).

  In case a search function cannot find a given encoding, it should return None.

**lookup (encoding)**

Looks up a codec tuple in the Python codec registry and returns the function tuple as defined above.

Encodings are first looked up in the registry’s cache. If not found, the list of registered search functions is scanned. If no codecs tuple is found, a **LookupError** is raised. Otherwise, the codecs tuple is stored in the cache and returned to the caller.

To simplify working with encoded files or stream, the module also defines these utility functions:

**open (filename, mode[, encoding[, errors[, buffering ]]])**

Open an encoded file using the given \(\text{mode}\) and return a wrapped version providing transparent encoding/decoding.

**Note:** The wrapped version will only accept the object format defined by the codecs, i.e. Unicode objects for most built-in codecs. Output is also codec-dependent and will usually be Unicode as well.

**encoding** specifies the encoding which is to be used for the the file.

**errors** may be given to define the error handling. It defaults to \(\text{‘strict’}\) which causes a **ValueError** to be raised in case an encoding error occurs.

**buffering** has the same meaning as for the built-in \(\text{open()}\) function. It defaults to line buffered.

**EncodedFile (file, input[, output[, errors ]])**

Return a wrapped version of file which provides transparent encoding translation.

Strings written to the wrapped file are interpreted according to the given \(\text{input}\) encoding and then written to the original file as strings using the \(\text{output}\) encoding. The intermediate encoding will usually be Unicode but depends on the specified codecs.

If **output** is not given, it defaults to **input**.

**errors** may be given to define the error handling. It defaults to \(\text{‘strict’},\) which causes **ValueError** to be raised in case an encoding error occurs.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

**BOM**
These constants define the byte order marks (BOM) used in data streams to indicate the byte order used in the stream or file. BOM is either BOM_BE or BOM_LE depending on the platform’s native byte order, while the others represent big endian (’BE’ suffix) and little endian (’LE’ suffix) byte order using 32-bit and 64-bit encodings.

See Also:

http://sourceforge.net/projects/python-codecs/

A SourceForge project working on additional support for Asian codecs for use with Python. They are in the early stages of development at the time of this writing — look in their FTP area for downloadable files.

4.8.1 Codec Base Classes

The codecs defines a set of base classes which define the interface and can also be used to easily write you own codecs for use in Python.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols.

The Codec class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the encode() and decode() methods may implement different error handling schemes by providing the errors string argument. The following string values are defined and implemented by all standard Python codecs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>’strict’</td>
<td>Raise ValueError (or a subclass); this is the default.</td>
</tr>
<tr>
<td>’ignore’</td>
<td>Ignore the character and continue with the next.</td>
</tr>
<tr>
<td>’replace’</td>
<td>Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER.</td>
</tr>
</tbody>
</table>

Codec Objects

The Codec class defines these methods which also define the function interfaces of the stateless encoder and decoder:

```
encode (input[, errors])
```

Encodes the object input and returns a tuple (output object, length consumed).

errors defines the error handling to apply. It defaults to ’strict’ handling.

The method may not store state in the Codec instance. Use StreamCodec for codecs which have to keep state in order to make encoding/decoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

```
decode (input[, errors])
```

Decodes the object input and returns a tuple (output object, length consumed).

input must be an object which provides the bf_getreadbuf buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.

errors defines the error handling to apply. It defaults to ’strict’ handling.
The method may not store state in the Codec instance. Use StreamCodec for codecs which have to keep state in order to make encoding/decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

The StreamWriter and StreamReader classes provide generic working interfaces which can be used to implement new encodings submodules very easily. See encodings.utf_8 for an example on how this is done.

StreamWriter Objects

The StreamWriter class is a subclass of Codec and defines the following methods which every stream writer must define in order to be compatible to the Python codec registry.

class StreamWriter(stream[, errors])

Constructor for a StreamWriter instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

stream must be a file-like object open for writing (binary) data.

The StreamWriter may implement different error handling schemes by providing the errors keyword argument. These parameters are defined:

- 'strict' Raise ValueError (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character

write(object)

Writes the object’s contents encoded to the stream.

writelines(list)

Writes the concatenated list of strings to the stream (possibly by reusing the write() method).

reset()

Flushes and resets the codec buffers used for keeping state.

Calling this method should ensure that the data on the output is put into a clean state, that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the StreamWriter must also inherit all other methods and attribute from the underlying stream.

StreamReader Objects

The StreamReader class is a subclass of Codec and defines the following methods which every stream reader must define in order to be compatible to the Python codec registry.

class StreamReader(stream[, errors])

Constructor for a StreamReader instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

stream must be a file-like object open for reading (binary) data.

The StreamReader may implement different error handling schemes by providing the errors keyword argument. These parameters are defined:

- 'strict' Raise ValueError (or a subclass); this is the default.
• 'ignore' Ignore the character and continue with the next.
• 'replace' Replace with a suitable replacement character.

`read(size)`
Decodes data from the stream and returns the resulting object.

`size` indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. `size` is intended to prevent having to decode huge files in one step.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

`readline([size])`
Read one line from the input stream and return the decoded data.

Note: Unlike the `readlines()` method, this method inherits the line breaking knowledge from the underlying stream's `readline()` method – there is currently no support for line breaking using the codec decoder due to lack of line buffering. Subclasses should however, if possible, try to implement this method using their own knowledge of line breaking.

`size`, if given, is passed as size argument to the stream’s `readline()` method.

`readlines([sizehint])`
Read all lines available on the input stream and return them as list of lines.

Line breaks are implemented using the codec’s decoder method and are included in the list entries.

`sizehint`, if given, is passed as `size` argument to the stream’s `read()` method.

`reset()`
Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attribute from the underlying stream.

The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.

**StreamReaderWriter Objects**

The `StreamReaderWriter` allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

```python
class StreamReaderWriter(stream, Reader, Writer, errors)
```
Creates a `StreamReaderWriter` instance. `stream` must be a file-like object. `Reader` and `Writer` must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attribute from the underlying stream.

**StreamRecoder Objects**

The `StreamRecoder` provide a frontend - backend view of encoding data which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.
class StreamRecoder(stream, encode, decode, Reader, Writer, errors)

Creates a StreamRecoder instance which implements a two-way conversion: `encode` and `decode` work on the frontend (the input to `read()` and output of `write()`) while `Reader` and `Writer` work on the backend (reading and writing to the stream).

You can use these objects to do transparent direct recodings from e.g. Latin-1 to UTF-8 and back.

`stream` must be a file-like object.

`encode`, `decode` must adhere to the Codec interface, `Reader`, `Writer` must be factory functions or classes providing objects of the the `StreamReader` and `StreamWriter` interface respectively.

`encode` and `decode` are needed for the frontend translation, `Reader` and `Writer` for the backend translation. The intermediate format used is determined by the two sets of codecs, e.g. the Unicode codecs will use Unicode as intermediate encoding.

Error handling is done in the same way as defined for the stream readers and writers.

StreamRecoder instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attribute from the underlying stream.

4.9 unicodedata — Unicode Database

This module provides access to the Unicode Character Database which defines character properties for all Unicode characters. The data in this database is based on the ‘UnicodeData.txt’ file version 3.0.0 which is publically available from ftp://ftp.unicode.org/.

The module uses the same names and symbols as defined by the UnicodeData File Format 3.0.0 (see http://www.unicode.org/Public/UNIDATA/UnicodeData.html). It defines the following functions:

`lookup(name)`
Look up character by name. If a character with the given name is found, return the corresponding Unicode character. If not found, `KeyError` is raised.

`name(unichr[, default])`
Returns the name assigned to the Unicode character `unichr` as a string. If no name is defined, `default` is returned, or, if not given, `ValueError` is raised.

`decimal(unichr[, default])`
Returns the decimal value assigned to the Unicode character `unichr` as integer. If no such value is defined, `default` is returned, or, if not given, `ValueError` is raised.

`digit(unichr[, default])`
Returns the digit value assigned to the Unicode character `unichr` as integer. If no such value is defined, `default` is returned, or, if not given, `ValueError` is raised.

`numeric(unichr[, default])`
Returns the numeric value assigned to the Unicode character `unichr` as float. If no such value is defined, `default` is returned, or, if not given, `ValueError` is raised.

`category(unichr)`
Returns the general category assigned to the Unicode character `unichr` as string.

`bidirectional(unichr)`
Returns the bidirectional category assigned to the Unicode character `unichr` as string. If no such value is defined, an empty string is returned.

`combining(unichr)`
Returns the canonical combining class assigned to the Unicode character `unichr` as integer. Returns 0 if no combining class is defined.
**mirrored**(unichr)

Returns the mirrored property of assigned to the Unicode character unichr as integer. Returns 1 if the character has been identified as a “mirrored” character in bidirectional text, 0 otherwise.

**decomposition**(unichr)

Returns the character decomposition mapping assigned to the Unicode character unichr as string. An empty string is returned in case no such mapping is defined.
CHAPTER
FIVE

miscellaneous services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here’s an overview:

- **doctest**
  A framework for verifying examples in docstrings.
- **unittest**
  Unit testing framework for Python.
- **math**
  Mathematical functions (sin() etc.).
- **cmath**
  Mathematical functions for complex numbers.
- **random**
  Generate pseudo-random numbers with various common distributions.
- **whrandom**
  Floating point pseudo-random number generator.
- **bisect**
  Array bisection algorithms for binary searching.
- **array**
  Efficient arrays of uniformly typed numeric values.
- **ConfigParser**
  Configuration file parser.
- **fileinput**
  Perl-like iteration over lines from multiple input streams, with “save in place” capability.
- **xreadlines**
  Efficient iteration over the lines of a file.
- **calendar**
  General functions for working with the calendar, including some emulation of the Unix **cal** program.
- **cmd**
  Build line-oriented command interpreters.
- **shlex**
  Simple lexical analysis for Unix shell-like languages.

5.1  **doctest** — Test docstrings represent reality

The **doctest** module searches a module’s docstrings for text that looks like an interactive Python session, then executes all such sessions to verify they still work exactly as shown. Here’s a complete but small example:
This is module example.

Example supplies one function, factorial. For example,

```python
>>> factorial(5)
120
```

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.
    If the result is small enough to fit in an int, return an int.
    Else return a long.
    ""
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
    ...  
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
    ...  
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(le100)
    Traceback (most recent call last):
    ...  
    OverflowError: n too large
    """
import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n:  # e.g., 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    try:
        result *= factor
    except OverflowError:
        result *= long(factor)
    factor += 1
return result

def _test():
    import doctest, example
    return doctest.testmod(example)

if __name__ == "__main__":
    _test()

If you run 'example.py' directly from the command line, doctest works its magic:

$ python example.py

There’s no output! That’s normal, and it means all the examples worked. Pass -v to the script, and doctest prints a detailed log of what it’s trying, and prints a summary at the end:

$ python example.py -v
Running example.__doc__
Trying: factorial(5)
Expecting: 120
  ok
0 of 1 examples failed in example.__doc__
Running example.factorial.__doc__
Trying: [factorial(n) for n in range(6)]
Expecting: [1, 1, 2, 6, 24, 120]
  ok
Trying: [factorial(long(n)) for n in range(6)]
Expecting: [1, 1, 2, 6, 24, 120]
  ok
Trying: factorial(30)
Expecting: 265252859812191058636308480000000L
  ok

And so on, eventually ending with:
Trying: factorial(1e100)
Expecting:
Traceback (most recent call last):
  ...
OverflowError: n too large
ok
0 of 8 examples failed in example.factorial.__doc__
2 items passed all tests:
  1 tests in example
  8 tests in example.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$

That’s all you need to know to start making productive use of doctest! Jump in. The docstrings in doctest.py contain detailed information about all aspects of doctest, and we’ll just cover the more important points here.

5.1.1 Normal Usage

In normal use, end each module $M$ with:

```python
def _test():
    import doctest, M  # replace M with your module’s name
    return doctest.testmod(M)  # ditto
if __name__ == "__main__":
    _test()
```

Then running the module as a script causes the examples in the docstrings to get executed and verified:

```python
python M.py
```

This won’t display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is ‘Test failed.’.

Run it with the `-v` switch instead:

```bash
python M.py -v
```

and a detailed report of all examples tried is printed to stdout, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=1` to testmod, or prohibit it by passing `verbose=0`. In either of those cases, `sys.argv` is not examined by testmod.

In any case, testmod returns a 2-tuple of ints $(f, t)$, where $f$ is the number of docstring examples that failed and $t$ is the total number of docstring examples attempted.
5.1.2 Which Docstrings Are Examined?

See ‘docstring.py’ for all the details. They’re unsurprising: the module docstring, and all function, class and method
docstrings are searched, with the exception of docstrings attached to objects with private names.

In addition, if $M.__test__$ exists and ”is true”, it must be a dict, and each entry maps a (string) name to a function
object, class object, or string. Function and class object docstrings found from $M.__test__$ are searched even if
the name is private, and strings are searched directly as if they were docstrings. In output, a key $K$ in $M.__test__$
appears with name

$$<\text{name of } M>.__test__.K$$

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.
While private names reached from $M$’s globals are skipped, all names reached from $M.__test__$ are searched.

5.1.3 What’s the Execution Context?

By default, each time testmod finds a docstring to test, it uses a copy of $M$’s globals, so that running tests on a module
doesn’t change the module’s real globals, and so that one test in $M$ can’t leave behind crumbs that accidentally allow
another test to work. This means examples can freely use any names defined at top-level in $M$, and names defined earlier
in the docstring being run. It also means that sloppy imports (see below) can cause examples in external docstrings to
use globals inappropriate for them.

You can force use of your own dict as the execution context by passing $\text{globs=your dict}$ to testmod() instead.
Presumably this would be a copy of $M.__dict__$ merged with the globals from other imported modules.

5.1.4 What About Exceptions?

No problem, as long as the only output generated by the example is the traceback itself. For example:

```python
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

Note that only the exception type and value are compared (specifically, only the last line in the traceback). The various
“File” lines in between can be left out (unless they add significantly to the documentation value of the example).

5.1.5 Advanced Usage

testmod() actually creates a local instance of class Tester, runs appropriate methods of that class, and merges
the results into global Tester instance master.

You can create your own instances of Tester, and so build your own policies, or even run methods of master
directly. See Tester.__doc__ for details.
5.1.6 How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine — just make sure the leading whitespace is rigidly consistent (you can mix tabs and spaces if you’re too lazy to do it right, but doctest is not in the business of guessing what you think a tab means).

```python
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...     print "NO!!"
...     print "NO!!"
...     no
NO
NO!!

>>> 
```

Any expected output must immediately follow the final ‘``` or ‘...’ line containing the code, and the expected output (if any) extends to the next ‘``` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output.
- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you need to double the backslash in the docstring version. This is simply because you’re in a string, and so the backslash must be escaped for it to survive intact. Like:

```python
>>> if "yes" == \"y\" + \"es\":
...     print 'yes'
yes
```

- The starting column doesn’t matter:

```python
>>> assert "Easy!"
>>> import math
>>> math.floor(1.9)
1.0
```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial ‘``` line that triggered it.
5.1.7 Warnings

1. Sloppy imports can cause trouble; e.g., if you do

   from XYZ import XYZclass

then XYZclass is a name in M.__dict__ too, and doctest has no way to know that XYZclass wasn’t defined in M. So it may try to execute the examples in XYZclass’s docstring, and those in turn may require a different set of globals to work correctly. I prefer to do “import *”-friendly imports, a la

   from XYZ import XYZclass as _XYZclass

and then the leading underscore makes _XYZclass a private name so testmod skips it by default. Other approaches are described in ‘doctest.py’.

2. doctest is serious about requiring exact matches in expected output. If even a single character doesn’t match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn’t guarantee about output. For example, when printing a dict, Python doesn’t guarantee that the key-value pairs will be printed in any particular order, so a test like

   >>> foo()  
   {"Hermione": "hippogryph", "Harry": "broomstick"}  
   >>>

is vulnerable! One workaround is to do

   >>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}  
   1  
   >>>

instead. Another is to do

   >>> d = foo().items()  
   >>> d.sort()  
   >>> d  
   [('Harry', 'broomstick'), ('Hermione', 'hippogryph')]

There are others, but you get the idea.
Another bad idea is to print things that embed an object address, like

   >>> id(1.0) # certain to fail some of the time  
   7948648  
   >>>

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.
>>> 1./7  # risky
0.14285714285714285
>>> print 1./7  # safer
0.14285714285714285
>>> print round(1./7, 6)  # much safer
0.142857

Numbers of the form $I/2.**J$ are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

>>> 3./4  # utterly safe
0.75

Simple fractions are also easier for people to understand, and that makes for better documentation.

### 5.1.8 Soapbox

The first word in doctest is ”doc”, and that’s why the author wrote doctest: to keep documentation up to date. It so happens that doctest makes a pleasant unit testing environment, but that’s not its primary purpose.

Choose docstring examples with care. There’s an art to this that needs to be learned — it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If possible, show just a few normal cases, show endcases, show interesting subtle cases, and show an example of each kind of exception that can be raised. You’re probably testing for endcases and subtle cases anyway in an interactive shell: doctest wants to make it as easy as possible to capture those sessions, and will verify they continue to work as designed forever after.

If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and ”things change”. I’m still amazed at how often one of my doctest examples stops working after a ”harmless” change.

For exhaustive testing, or testing boring cases that add no value to the docs, define a `__test__` dict instead. That’s what it’s for.

### 5.2 `unittest` — Unit testing framework

The Python unit testing framework, often referred to as “PyUnit,” is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent’s Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

PyUnit supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The `unittest` module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, PyUnit supports some important concepts:

**test fixture**

A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

**test case**

A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. PyUnit provides a base class, `TestCase`, which may be used to create new test cases.
test suite
A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner
A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

The test case and test fixture concepts are supported through the `TestCase` and `FunctionTestCase` classes; the former should be used when creating new tests, and the later can be used when integrating existing test code with a PyUnit-driven framework. When building test fixtures using `TestCase`, the `setUp()` and `tearDown()` methods can be overridden to provide initialization and cleanup for the fixture. With `FunctionTestCase`, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the `TestCase` will only be used to run a single test method, so a new fixture is created for each test.

Test suites are implemented by the `TestSuite` class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in “child” test suites are run.

A test runner is an object that provides a single method, `run()`, which accepts a `TestCase` or `TestSuite` object as a parameter, and returns a result object. The class `TestResult` is provided for use as the result object. PyUnit provide the `TextTestRunner` as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

See Also:

  The source for further information on PyUnit.

  Kent Beck’s original paper on testing frameworks using the pattern shared by `unittest`.

5.2.1 Organizing test code

The basic building blocks of unit testing are test cases — single scenarios that must be set up and checked for correctness. In PyUnit, test cases are represented by instances of the `TestCase` class in the `unittest` module. To make your own test cases you must write subclasses of `TestCase`, or use `FunctionTestCase`.

An instance of a `TestCase`-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest test case subclass will simply override the `runTest()` method in order to perform specific testing code:

```python
import unittest
class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget("The widget")
        self.failUnless(widget.size() == (50,50), 'incorrect default size')
```
Note that in order to test something, we use the one of the assert*() or fail*() methods provided by the 
TestCase base class. If the test fails when the test case runs, an exception will be raised, and the testing framework 
will identify the test case as a failure. Other exceptions that do not arise from checks made through the assert*() 
and fail*() methods are identified by the testing framework as dfnerrors.

The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we 
call its constructor without arguments:

    testCase = DefaultWidgetSizeTestCase()

Now, such test cases can be numerous, and their set-up can be repetitive. In the above case, constructing a “Widget” 
in each of 100 Widget test case subclasses would mean unsightly duplication.

Luckily, we can factor out such set-up code by implementing a method called setUp(), which the testing framework 
will automatically call for us when we run the test:

```python
import unittest
class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.failUnless(self.widget.size() == (50,50),
                        'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150),
                        'wrong size after resize')
```

If the setUp() method raises an exception while the test is running, the framework will consider the test to have 
suffered an error, and the runTest() method will not be executed.

Similarly, we can provide a tearDown() method that tidies up after the runTest() method has been run:

```python
import unittest
class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

If setUp() succeeded, the tearDown() method will be run regardless of whether or not runTest() succeeded. 
Such a working environment for the testing code is called a fixture.

Often, many small test cases will use the same fixture. In this case, we would end up subclassing SimpleWid-
getWidgetTestCase into many small one-method classes such as DefaultWidgetSizeTestCase. This is time-
consuming and discouraging, so in the same vein as JUnit, PyUnit provides a simpler mechanism:
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testDefaultSize(self):
        self.failUnless(self.widget.size() == (50,50), 'incorrect default size')

    def testResize(self):
        self.widget.resize(100,150)
        self.failUnless(self.widget.size() == (100,150), 'wrong size after resize')

Here we have not provided a runTest() method, but have instead provided two different test methods. Class
instances will now each run one of the test*() methods, with self.widget created and destroyed separately
for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the
method name in the constructor:

defaultSizeTestCase = WidgetTestCase("testDefaultSize")
resizeTestCase = WidgetTestCase("testResize")

Test case instances are grouped together according to the features they test. PyUnit provides a mechanism for this: the
test suite, represented by the class TestSuite in the unittest module:

widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase("testDefaultSize"))
widgetTestSuite.addTest(WidgetTestCase("testResize"))

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that
returns a pre-built test suite:

def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testDefaultSize"))
    suite.addTest(WidgetTestCase("testResize"))
    return suite

or even:

class WidgetTestSuite(unittest.TestSuite):
    def __init__(self):
        unittest.TestSuite.__init__(self,map(WidgetTestCase,
                                            {"testDefaultSize",
                                             "testResize"}))

5.2. unittest — Unit testing framework
Since it is a common pattern to create a `TestCase` subclass with many similarly named test functions, there is a convenience function called `makeSuite()` provided in the `unittest` module that constructs a test suite that comprises all of the test cases in a test case class:

```python
suite = unittest.makeSuite(WidgetTestCase,'test')
```

Note that when using the `makeSuite()` function, the order in which the various test cases will be run by the test suite is the order determined by sorting the test function names using the `cmp()` built-in function.

Often it is desirable to group suites of test cases together, so as to run tests for the whole system at once. This is easy, since `TestSuite` instances can be added to a `TestSuite` just as `TestCase` instances can be added to a `TestSuite`:

```python
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite((suite1, suite2))
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (e.g. `'widget.py'`), but there are several advantages to placing the test code in a separate module, such as `'widgettests.py'`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code if tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

### 5.2.2 Re-using old test code

Some users will find that they have existing test code that they would like to run from PyUnit, without converting every old test function to a `TestCase` subclass.

For this reason, PyUnit provides a `FunctionTestCase` class. This subclass of `TestCase` can be used to wrap an existing test function. Set-up and tear-down functions can also optionally be wrapped.

Given the following test function:

```python
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:
testcase = unittest.FunctionTestCase(testSomething)

If there are additional set-up and tear-down methods that should be called as part of the test case’s operation, they can also be provided:

testcase = unittest.FunctionTestCase(testSomething, 
    setUp=makeSomethingDB, 
    tearDown=deleteSomethingDB)

Note: PyUnit supports the use of AssertionError as an indicator of test failure, but does not recommend it. Future versions may treat AssertionError differently.

5.2.3 Classes and functions

class TestCase()

Instances of the TestCase class represent the smallest testable units in a set of tests. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the test, and methods that the test code can use to check for and report various kinds of failures.

class FunctionTestCase(testFunc[, setUp[, tearDown[, description]]]])

This class implements the portion of the TestCase interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a unittest-based test framework.

class TestSuite([tests])

This class represents an aggregation of individual tests cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case, but all the contained tests and test suites are executed. Additional methods are provided to add test cases and suites to the aggregation. If tests is given, it must be a sequence of individual tests that will be added to the suite.

class TestLoader()

This class is responsible for loading tests according to various criteria and returning them wrapped in a TestSuite. It can load all tests within a given module or TestCase class. When loading from a module, it considers all TestCase-derived classes. For each such class, it creates an instance for each method with a name beginning with the string ‘test’.

defaultTestLoader

Instance of the TestLoader class which can be shared. If no customization of the TestLoader is needed, this instance can always be used instead of creating new instances.

class TextTestRunner([stream[, descriptions[, verbosity]]]])

A basic test runner implementation which prints results on standard output. It has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.

main([module[, defaultTest[, argv[, testRunner[, testRunner]]]]]])

A command-line program that runs a set of tests; this is primarily for making test modules conveniently executable. The simplest use for this function is:

    if __name__ == '__main__':
        unittest.main()
5.2.4 Test Case Objects

Each `TestCase` instance represents a single test, but each concrete subclass may be used to define multiple tests — the concrete class represents a single test fixture. The fixture is created and cleaned up for each test case.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group are:

- `setUp()`:
  Method called to prepare the test fixture. This is called immediately before calling the test method; any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

- `tearDown()`:
  Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception raised by this method will be considered an error rather than a test failure. This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

- `run([result])`:
  Run the test, collecting the result into the test result object passed as `result`. If `result` is omitted or `None`, a temporary result object is created and used, but is not made available to the caller. This is equivalent to simply calling the `TestCase` instance.

- `debug()`:
  Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The test code can use any of the following methods to check for and report failures.

- `assert_(expr[, msg])`:
  Signal a test failure if `expr` is false; the explanation for the error will be `msg` if given, otherwise it will be `None`.

- `failUnless(expr[, msg])`:
  Test that `expr` is true. If `expr` is false, the test will fail with the explanation given by `msg`, or `None`. Note that using `failUnless()` improves upon doing the comparison as the default value for `msg` can be computed to include representations of both `first` and `second`.

- `assertEqual(first, second[, msg])`:
  Test that `first` and `second` are equal. If the values do not compare equal, the test will fail with the explanation given by `msg`, or `None`. Note that using `failUnlessEqual()` improves upon doing the comparison as the first parameter to `failUnless()` is that the default value for `msg` can be computed to include representations of both `first` and `second`.

- `failIfEqual(first, second[, msg])`:
  Test that `first` and `second` are not equal. If the values do compare equal, the test will fail with the explanation given by `msg`, or `None`. Note that using `failIfEqual()` improves upon doing the comparison as the first parameter to `failUnless()` is that the default value for `msg` can be computed to include representations of both `first` and `second`.

- `assertRaises(exception, callable, ...)`:
  Test that an exception is raised when `callable` is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if `exception` is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as `exception`.

- `failIf(expr[, msg])`
The inverse of the `failUnless()` method is the `failIf()` method. This signals a test failure if `expr` is true, with `msg` or `None` for the error message.

```python
fail([msg])
```
Signals a test failure unconditionally, with `msg` or `None` for the error message.

### failureException
This class attribute gives the exception raised by the `test()` method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is `AssertionError`.

Testing frameworks can use the following methods to collect information on the test:

```python
countTestCases()
```
Return the number of tests represented by the this test object. For `TestCase` instances, this will always be 1, but this method is also implemented by the `TestSuite` class, which can return larger values.

```python
defaultTestResult()
```
Return the default type of test result object to be used to run this test.

```python
id()
```
Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class names.

```python
shortDescription()
```
Returns a one-line description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method’s docstring, if available, or `None`.

### 5.2.5 TestSuite Objects

`TestSuite` objects behave much like `TestCase` objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups that should be run together. Some additional methods are available to add tests to `TestSuite` instances:

```python
addTest(test)
```
Add a `TestCase` or `TestSuite` to the set of tests that make up the suite.

```python
addTests(tests)
```
Add all the tests from a sequence of `TestCase` and `TestSuite` instances to this test suite.

### 5.2.6 TestResult Objects

A `TestResult` object stores the results of a set of tests. The `TestCase` and `TestSuite` classes ensure that results are properly stored; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of `unittest` may want access to the `TestResult` object generated by running a set of tests for reporting purposes; a `TestResult` instance is returned by the `TestRunner.run()` method for this purpose.

Each instance holds the total number of tests run, and collections of failures and errors that occurred among those test runs. The collections contain tuples of `(testCase, exceptioninfo)`, where `exceptioninfo` is a tuple as returned by `sys.exc_info()`.

`TestResult` instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

```python
errors
```
A list containing pairs of `TestCase` instances and the `sys.exc_info()` results for tests which raised an exception but did not signal a test failure.
failures
A list containing pairs of TestCase instances and the sys.exc_info() results for tests which signalled a
c Failure in the code under test.

testsRun
The number of tests which have been started.

wasSuccessful()
Returns true if all tests run so far have passed, otherwise returns false.

The following methods of the TestResult class are used to maintain the internal data structures, and may be
extended in subclasses to support additional reporting requirements. This is particularly useful in building GUI tools
which support interactive reporting while tests are being run.

startTest(test)
Called when the test case test is about to be run.

stopTest(test)
Called when the test case test has been executed, regardless of the outcome.

addError(test, err)
Called when the test case test raises an exception without signalling a test failure. err is a tuple of the form
returned by sys.exc_info(): (type, value, traceback).

addFailure(test, err)
Called when the test case test signals a failure. err is a tuple of the form returned by sys.exc_info():
(type, value, traceback).

addSuccess(test)
This method is called for a test that does not fail; test is the test case object.

One additional method is available for TestResult objects:

stop()
This method can be called to signal that the set of tests being run should be aborted. Once this has been
called, the TestRunner object return to its caller without running any additional tests. This is used by the
TextTestRunner class to stop the test framework when the user signals an interrupt from the keyboard. GUI
tools which provide runners can use this in a similar manner.

5.2.7 TestLoader Objects

The TestLoader class is used to create test suites from classes and modules. Normally, there is no need to create an
instance of this class; the unittest module provides an instance that can be shared as the defaultTestLoader
module attribute. Using a subclass or instance would allow customization of some configurable properties.

TestLoader objects have the following methods:

loadTestsFromTestCase(testCaseClass)
Return a suite of all tests cases contained in the TestCase-derived class testCaseClass.

loadTestsFromModule(module)
Return a suite of all tests cases contained in the given module. This method searches module for classes derived
from TestCase and creates an instance of the class for each test method defined for the class.

Warning: While using a hierarchy of Testcase-derived classes can be convenient in sharing fixtures and
helper functions, defining test methods on base classes that are not intended to be instantiated directly does not
play well with this method. Doing so, however, can be useful when the fixtures are different and defined in
subclasses.

loadTestsFromName(name[, module])
Return a suite of all tests cases given a string specifier.
The specifier *name* may resolve either to a module, a test case class, a test method within a test case class, or a callable object which returns a `TestCase` or `TestSuite` instance.

The method optionally resolves *name* relative to a given module.

```python
loadTestsFromNames(names[, module])
```

Similar to `loadTestsFromName()`, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

```python
getTestCaseNames(testCaseClass)
```

Return a sorted sequence of method names found within `testCaseClass`.

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

- **testMethodPrefix**  
  String giving the prefix of method names which will be interpreted as test methods. The default value is ‘test’.

- **sortTestMethodsUsing**  
  Function to be used to compare method names when sorting them in `getTestCaseNames()`. The default value is the built-in `cmp()` function; it can be set to `None` to disable the sort.

- **suiteClass**  
  Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the `TestSuite` class.

### 5.3 math — Mathematical functions

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don’t is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions provided by this module:

- **acos**(*x*)  
  Return the arc cosine of *x*.

- **asin**(*x*)  
  Return the arc sine of *x*.

- **atan**(*x*)  
  Return the arc tangent of *x*.

- **atan2**(*y, x*)  
  Return \( \tan^{-1}(y / x) \).

- **ceil**(*x*)  
  Return the ceiling of *x* as a float.

- **cos**(*x*)  
  Return the cosine of *x*.

- **cosh**(*x*)  
  Return the hyperbolic cosine of *x*.

- **exp**(*x*)
Return $e^{**x}$.

**fabs** $(x)$  
Return the absolute value of the floating point number $x$.

**floor** $(x)$  
Return the floor of $x$ as a float.

**fmod** $(x, y)$  
Return $fmod(x, y)$, as defined by the platform C library. Note that the Python expression $x \% y$ may not return the same result.

**frexp** $(x)$  
Return the mantissa and exponent of $x$ as the pair $(m, e)$. $m$ is a float and $e$ is an integer such that $x == m * 2**e$. If $x$ is zero, returns $(0.0, 0)$, otherwise $0.5 <= abs(m) < 1$.

**hypot** $(x, y)$  
Return the Euclidean distance, $\sqrt{x^2 + y^2}$.

**ldexp** $(x, i)$  
Return $x * (2**i)$.

**log** $(x)$  
Return the natural logarithm of $x$.

**log10** $(x)$  
Return the base-10 logarithm of $x$.

**modf** $(x)$  
Return the fractional and integer parts of $x$. Both results carry the sign of $x$. The integer part is returned as a float.

**pow** $(x, y)$  
Return $x**y$.

**sin** $(x)$  
Return the sine of $x$.

**sinh** $(x)$  
Return the hyperbolic sine of $x$.

**sqrt** $(x)$  
Return the square root of $x$.

**tan** $(x)$  
Return the tangent of $x$.

**tanh** $(x)$  
Return the hyperbolic tangent of $x$.

Note that **frexp()** and **modf()** have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an ‘output parameter’ (there is no such thing in Python).

The module also defines two mathematical constants:

**pi**  
The mathematical constant $\pi$.

**e**  
The mathematical constant $e$.

See Also:

Module **cmath** (section 5.4):
Complex number versions of many of these functions.

5.4 `cmath` — Mathematical functions for complex numbers

This module is always available. It provides access to mathematical functions for complex numbers. The functions are:

- `acos(x)`
  - Return the arc cosine of `x`.

- `acosh(x)`
  - Return the hyperbolic arc cosine of `x`.

- `asin(x)`
  - Return the arc sine of `x`.

- `asinh(x)`
  - Return the hyperbolic arc sine of `x`.

- `atan(x)`
  - Return the arc tangent of `x`.

- `atanh(x)`
  - Return the hyperbolic arc tangent of `x`.

- `cos(x)`
  - Return the cosine of `x`.

- `cosh(x)`
  - Return the hyperbolic cosine of `x`.

- `exp(x)`
  - Return the exponential value $e^x$.

- `log(x)`
  - Return the natural logarithm of `x`.

- `log10(x)`
  - Return the base-10 logarithm of `x`.

- `sin(x)`
  - Return the sine of `x`.

- `sinh(x)`
  - Return the hyperbolic sine of `x`.

- `sqrt(x)`
  - Return the square root of `x`.

- `tan(x)`
  - Return the tangent of `x`.

- `tanh(x)`
  - Return the hyperbolic tangent of `x`.

The module also defines two mathematical constants:

- `pi`
  - The mathematical constant $\pi$, as a real.

- `e`
  - The mathematical constant $e$, as a real.
Note that the selection of functions is similar, but not identical, to that in module `math`. The reason for having two modules is that some users aren’t interested in complex numbers, and perhaps don’t even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in `cmath` always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

5.5 `random` — Generate pseudo-random numbers

This module implements pseudo-random number generators for various distributions. For integers, uniform selection from a range. For sequences, uniform selection of a random element, and a function to generate a random permutation of a list in-place. On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distribution of angles, the circular uniform and von Mises distributions are available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range `[0.0, 1.0)`. Python uses the standard Wichmann-Hill generator, combining three pure multiplicative congruential generators of modulus 30269, 30307 and 30323. Its period (how many numbers it generates before repeating the sequence exactly) is 6,953,607,871,644. While of much higher quality than the `rand()` function supplied by most C libraries, the theoretical properties are much the same as for a single linear congruential generator of large modulus. It is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions in this module are not thread-safe: if you want to call these functions from multiple threads, you should explicitly serialize the calls. Else, because no critical sections are implemented internally, calls from different threads may see the same return values.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don’t share state. This is especially useful for multi-threaded programs, creating a different instance of `Random` for each thread, and using the `jumpahead()` method to ensure that the generated sequences seen by each thread don’t overlap (see example below).

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the `random()`, `seed()`, `getstate()`, `setstate()` and `jumpahead()` methods.

Here’s one way to create threadsafe distinct and non-overlapping generators:

```python
def create_generators(num, delta, firstseed=None):
    """Return list of num distinct generators.
    Each generator has its own unique segment of delta elements from Random.random()'s full period.
    Seed the first generator with optional arg firstseed (default is None, to seed from current time).
    """
    from random import Random
    g = Random(firstseed)
    result = [g]
    for i in range(num - 1):
        laststate = g.getstate()
        g = Random()
        g.setstate(laststate)
        g.jumpahead(delta)
        result.append(g)
    return result

gens = create_generators(10, 1000000)
```
That creates 10 distinct generators, which can be passed out to 10 distinct threads. The generators don’t share state so can be called safely in parallel. So long as no thread calls its g.random() more than a million times (the second argument to create_generators(), the sequences seen by each thread will not overlap. The period of the underlying Wichmann-Hill generator limits how far this technique can be pushed.

Just for fun, note that since we know the period, jumpahead() can also be used to “move backward in time:”

```python
>>> g = Random(42)  # arbitrary
>>> g.random()
0.25420336316883324
>>> g.jumpahead(6953607871644L - 1)  # move *back* one
>>> g.random()
0.25420336316883324
```

Bookkeeping functions:

**seed**(x)

Initialize the basic random number generator. Optional argument x can be any hashable object. If `(x)` is omitted or None, current system time is used; current system time is also used to initialize the generator when the module is first imported. If `(x)` is not None or an int or long, hash(x) is used instead. If `x` is an int or long, `x` is used directly. Distinct values between 0 and 27814431486575L inclusive are guaranteed to yield distinct internal states (this guarantee is specific to the default Wichmann-Hill generator, and may not apply to subclasses supplying their own basic generator).

**whseed**(x)

This is obsolete, supplied for bit-level compatibility with versions of Python prior to 2.1. See seed for details. whseed does not guarantee that distinct integer arguments yield distinct internal states, and can yield no more than about 2**24 distinct internal states in all.

**getstate**( )

Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state. New in version 2.1.

**setstate**(state)

`state` should have been obtained from a previous call to getstate(), and setstate() restores the internal state of the generator to what it was at the time setstate() was called. New in version 2.1.

**jumpahead**(n)

Change the internal state to what it would be if random() were called n times, but do so quickly. n is a non-negative integer. This is most useful in multi-threaded programs, in conjunction with multiple instances of the Random class: setstate() or seed() can be used to force all instances into the same internal state, and then jumpahead() can be used to force the instances’ states as far apart as you like (up to the period of the generator). New in version 2.1.

Functions for integers:

**randrange**(start, stop[, step])

Return a randomly selected element from range(start, stop, step). This is equivalent to choice(range(start, stop, step)), but doesn’t actually build a range object. New in version 1.5.2.

**randint**(a, b)

Deprecated since release 2.0. Use randrange() instead.

Return a random integer N such that a <= N <= b.

Functions for sequences:

**choice**(seq)

Return a random element from the non-empty sequence seq.
*shuffle*(x[, random])

Shuffle the sequence x in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function *random()*.

Note that for even rather small `len(x)`, the total number of permutations of x is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution’s equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

*random()*

Return the next random floating point number in the range [0.0, 1.0).

*uniform(a, b)*

Return a random real number *N* such that *a* <= *N* < *b*.

*betavariate(alpha, beta)*

Beta distribution. Conditions on the parameters are *alpha* > -1 and *beta* > -1. Returned values range between 0 and 1.

*cunifvariate(mean, arc)*

Circular uniform distribution. *mean* is the mean angle, and *arc* is the range of the distribution, centered around the mean angle. Both values must be expressed in radians, and can range between 0 and pi. Returned values range between *mean* - *arc*/2 and *mean* + *arc*/2.

*expovariate(lambd)*

Exponential distribution. *lambd* is 1.0 divided by the desired mean. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity.

*gamma(alpha, beta)*

Gamma distribution. (Not the gamma function!) Conditions on the parameters are *alpha* > -1 and *beta* > 0.

*gauss(mu, sigma)*

Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the *normalvariate()* function defined below.

*lognormvariate(mu, sigma)*

Log normal distribution. If you take the natural logarithm of this distribution, you’ll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

*normalvariate(mu, sigma)*

Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

*vonmisesvariate(mu, kappa)*

*mu* is the mean angle, expressed in radians between 0 and 2*pi, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2*pi.

*paretovariate(alpha)*

Pareto distribution. *alpha* is the shape parameter.

*weibullvariate(alpha, beta)*

Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

See Also:

5.6 whrandom — Pseudo-random number generator

Deprecated since release 2.1. Use random instead.

Note: This module was an implementation detail of the random module in releases of Python prior to 2.1. It is no longer used. Please do not use this module directly; use random instead.

This module implements a Wichmann-Hill pseudo-random number generator class that is also named whrandom. Instances of the whrandom class conform to the Random Number Generator interface described in section ?? . They also offer the following method, specific to the Wichmann-Hill algorithm:

\[ \text{seed}([x, y, z]) \]

Initializes the random number generator from the integers \( x, y \) and \( z \). When the module is first imported, the random number is initialized using values derived from the current time. If \( x, y, \) and \( z \) are either omitted or 0, the seed will be computed from the current system time. If one or two of the parameters are 0, but not all three, the zero values are replaced by ones. This causes some apparently different seeds to be equal, with the corresponding result on the pseudo-random series produced by the generator.

\[ \text{choice}(\text{seq}) \]

Chooses a random element from the non-empty sequence \( \text{seq} \) and returns it.

\[ \text{randint}(a, b) \]

Returns a random integer \( N \) such that \( a \leq N \leq b \).

\[ \text{random}() \]

Returns the next random floating point number in the range \([0.0 \ldots 1.0)\).

\[ \text{seed}(x, y, z) \]

Initializes the random number generator from the integers \( x, y \) and \( z \). When the module is first imported, the random number is initialized using values derived from the current time.

\[ \text{uniform}(a, b) \]

Returns a random real number \( N \) such that \( a \leq N < b \).

When imported, the whrandom module also creates an instance of the whrandom class, and makes the methods of that instance available at the module level. Therefore one can write either \( N = \text{whrandom.random}() \) or:

```python
generator = whrandom.whrandom()
N = generator.random()
```

Note that using separate instances of the generator leads to independent sequences of pseudo-random numbers.

See Also:

Module random (section 5.5):
Generators for various random distributions and documentation for the Random Number Generator interface.


5.7 bisect — Array bisection algorithm

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called bisect because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (i.e., the boundary conditions are already right!).

The following functions are provided:
bisect_left (list, item[, lo[, hi]])
Locate the proper insertion point for item in list to maintain sorted order. The parameters lo and hi may be used to specify a subset of the list which should be considered; by default the entire list is used. If item is already present in list, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to list.insert(). This assumes that list is already sorted. New in version 2.1.

bisect_right (list, item[, lo[, hi]])
Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing entries of item in list. New in version 2.1.

bisect (...)
Alias for bisect_right().

insort_left (list, item[, lo[, hi]])
Insert item in list in sorted order. This is equivalent to list.insert(bisect.bisect_left(list, item, lo, hi), item). This assumes that list is already sorted. New in version 2.1.

insort_right (list, item[, lo[, hi]])
Similar to insort_left(), but inserting item in list after any existing entries of item. New in version 2.1.

insort (...)
Alias for insort_right().

5.7.1 Example

The bisect() function is generally useful for categorizing numeric data. This example uses bisect() to look up a letter grade for an exam total (say) based on a set of ordered numeric breakpoints: 85 and up is an ‘A’, 75..84 is a ‘B’, etc.

```python
>>> grades = "FEDCBA"
>>> breakpoints = [30, 44, 66, 75, 85]
>>> from bisect import bisect
>>> def grade(total):
...     return grades[bisect(breakpoints, total)]
...     ...
>>> grade(66)
'C'
>>> map(grade, [33, 99, 77, 44, 12, 88])
['E', 'A', 'B', 'D', 'F', 'A']
```

5.8 array — Efficient arrays of numeric values

This module defines a new object type which can efficiently represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:
## 5.8. `array` — Efficient arrays of numeric values

<table>
<thead>
<tr>
<th>Type code</th>
<th>C Type</th>
<th>Minimum size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>'b'</td>
<td>signed int</td>
<td>1</td>
</tr>
<tr>
<td>'B'</td>
<td>unsigned int</td>
<td>1</td>
</tr>
<tr>
<td>'h'</td>
<td>signed int</td>
<td>2</td>
</tr>
<tr>
<td>'H'</td>
<td>unsigned int</td>
<td>2</td>
</tr>
<tr>
<td>'i'</td>
<td>signed int</td>
<td>2</td>
</tr>
<tr>
<td>'I'</td>
<td>unsigned int</td>
<td>2</td>
</tr>
<tr>
<td>'l'</td>
<td>signed int</td>
<td>4</td>
</tr>
<tr>
<td>'L'</td>
<td>unsigned int</td>
<td>4</td>
</tr>
<tr>
<td>'f'</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>'d'</td>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the `itemsize` attribute. The values stored for 'L' and 'I' items will be represented as Python long integers when retrieved, because Python's plain integer type cannot represent the full range of C's unsigned (long) integers.

The module defines the following function and type object:

```python
array(typecode[, initializer])
```

Return a new array whose items are restricted by `typecode`, and initialized from the optional `initializer` value, which must be a list or a string. The list or string is passed to the new array’s `fromlist()` or `fromstring()` method (see below) to add initial items to the array.

**ArrayType**

Type object corresponding to the objects returned by `array()`.

Array objects support the following data items and methods:

- `typecode`
  - The typecode character used to create the array.
- `itemsize`
  - The length in bytes of one array item in the internal representation.
- `append(x)`
  - Append a new item with value `x` to the end of the array.
- `buffer_info()`
  - Return a tuple `(address, length)` giving the current memory address and the length in bytes of the buffer used to hold array’s contents. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.
- `byteswap()`
  - “Byteswap” all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, `RuntimeError` is raised. It is useful when reading data from a file written on a machine with a different byte order.
- `count(x)`
  - Return the number of occurrences of `x` in the array.
- `extend(a)`
  - Append array items from `a` to the end of the array.
- `fromfile(f, n)`
  - Read `n` items (as machine values) from the file object `f` and append them to the end of the array. If less than `n` items are available, `EOFError` is raised, but the items that were available are still inserted into the array. `f` must be a real built-in file object; something else with a `read()` method won’t do.
fromlist (list)
    Append items from the list. This is equivalent to ‘for x in list: a.append(x)’ except that if there is a type error, the array is unchanged.

fromstring (s)
    Appends items from the string, interpreting the string as an array of machine values (i.e. as if it had been read from a file using the fromfile() method).

index (x)
    Return the smallest i such that i is the index of the first occurrence of x in the array.

insert (i, x)
    Insert a new item with value x in the array before position i.

pop ([i])
    Removes the item with the index i from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.

read (f, n)
    Deprecated since release 1.5.1. Use the fromfile() method.
    Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won’t do.

remove (x)
    Remove the first occurrence of x from the array.

reverse ()
    Reverse the order of the items in the array.

tofile (f)
    Write all items (as machine values) to the file object f.

tolist ()
    Convert the array to an ordinary list with the same items.

tostring ()
    Convert the array to an array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the tofile() method.)

write (f)
    Deprecated since release 1.5.1. Use the tofile() method.
    Write all items (as machine values) to the file object f.

When an array object is printed or converted to a string, it is represented as array(typecode, initializer). The initializer is omitted if the array is empty, otherwise it is a string if the typecode is ’c’, otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using reverse quotes (’ ’), so long as the array() function has been imported using ‘from array import array’. Examples:

    array(’l’)
    array(’c’, ’hello world’)
    array(’l’, [1, 2, 3, 4, 5])
    array(’d’, [1.0, 2.0, 3.14])

See Also:
Module struct (section 4.3):
    Packing and unpacking of heterogeneous binary data.
Module xdrlib (section 12.9):
Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.


5.9 ConfigParser — Configuration file parser

This module defines the class ConfigParser. The ConfigParser class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

The configuration file consists of sections, lead by a ‘[section]’ header and followed by ‘name: value’ entries, with continuations in the style of RFC 822; ‘name=value’ is also accepted. Note that leading whitespace is removed from values. The optional values can contain format strings which refer to other values in the same section, or values in a special DEFAULT section. Additional defaults can be provided upon initialization and retrieval. Lines beginning with ‘#’ or ‘;’ are ignored and may be used to provide comments.

For example:

```
foodir: %(dir)s/whatever
dir=frob
```

would resolve the ‘%(dir)s’ to the value of ‘dir’ (‘frob’ in this case). All reference expansions are done on demand.

Default values can be specified by passing them into the ConfigParser constructor as a dictionary. Additional defaults may be passed into the get() method which will override all others.

```
class ConfigParser([defaults])
```

Return a new instance of the ConfigParser class. When defaults is given, it is initialized into the dictionary of intrinsic defaults. The keys must be strings, and the values must be appropriate for the ‘%()s’ string interpolation. Note that __name__ is an intrinsic default; its value is the section name, and will override any value provided in defaults.

exception NoSectionError
Exception raised when a specified section is not found.

exception DuplicateSectionError
Exception raised when multiple sections with the same name are found, or if add_section() is called with the name of a section that is already present.

exception NoOptionError
Exception raised when a specified option is not found in the specified section.

exception InterpolationError
Exception raised when problems occur performing string interpolation.

exception InterpolationDepthError
Exception raised when string interpolation cannot be completed because the number of iterations exceeds MAX_INTERPOLATION_DEPTH.

exception MissingSectionHeaderError
Exception raised when attempting to parse a file which has no section headers.

**exception ParsingError**
Exception raised when errors occur attempting to parse a file.

**MAX_INTERPOLATION_DEPTH**
The maximum depth for recursive interpolation for `get()` when the `raw` parameter is false. Setting this does not change the allowed recursion depth.

See Also:
*Module shlex (section 5.14):*
Support for creating UNIX shell-like mini-languages which can be used as an alternate format for application configuration files.

### 5.9.1 ConfigParser Objects

ConfigParser instances have the following methods:

**defaults()**
Return a dictionary containing the instance-wide defaults.

**sections()**
Return a list of the sections available; DEFAULT is not included in the list.

**add_section(section)**
Add a section named `section` to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised.

**has_section(section)**
Indicates whether the named section is present in the configuration. The DEFAULT section is not acknowledged.

**options(section)**
Returns a list of options available in the specified `section`.

**has_option(section, option)**
If the given section exists, and contains the given option, return 1; otherwise return 0. (New in 1.6)

**read(filenames)**
Read and parse a list of filenames. If `filenames` is a string or Unicode string, it is treated as a single filename.

**readfp(fp[, filename])**
Read and parse configuration data from the file or file-like object in `fp` (only the `readline()` method is used). If `filename` is omitted and `fp` has a `name` attribute, that is used for `filename`; the default is ‘<???>’.

**get(section, option[, raw[, vars]])**
Get an `option` value for the provided `section`. All the ‘%’ interpolations are expanded in the return values, based on the defaults passed into the constructor, as well as the options `vars` provided, unless the `raw` argument is true.

**getint(section, option)**
A convenience method which coerces the `option` in the specified `section` to an integer.

**getfloat(section, option)**
A convenience method which coerces the `option` in the specified `section` to a floating point number.

**getboolean(section, option)**
A convenience method which coerces the `option` in the specified `section` to a boolean value. Note that the only accepted values for the option are ‘0’ and ‘1’, any others will raise `ValueError`.

**set(section, option, value)**
If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. (New in 1.6)
write (fileobject)
Write a representation of the configuration to the specified file object. This representation can be parsed by a
future read() call. (New in 1.6)

remove_option (section, option)
Remove the specified option from the specified section. If the section does not exist, raise NoSectionError. If
the option existed to be removed, return 1; otherwise return 0. (New in 1.6)

remove_section (section)
Remove the specified section from the configuration. If the section in fact existed, return 1. Otherwise return 0.

optionxform (option)
Transforms the option name option as found in an input file or as passed in by client code to the form that should
be used in the internal structures. The default implementation returns a lower-case version of option; subclasses
may override this or client code can set an attribute of this name on instances to affect this behavior. Setting this
to str(), for example, would make option names case sensitive.

5.10 fileinput — Iterate over lines from multiple input streams

This module implements a helper class and functions to quickly write a loop over standard input or a list of files.
The typical use is:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If
a filename is ‘-‘, it is also replaced by sys.stdin. To specify an alternative list of filenames, pass it as the first
argument to input(). A single file name is also allowed.

All files are opened in text mode. If an I/O error occurs during opening or reading a file, IOError is raised.
If sys.stdin is used more than once, the second and further use will return no lines, except perhaps for interactive
use, or if it has been explicitly reset (e.g. using sys.stdin.seek(0)).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all
is when the last file opened is empty.

It is possible that the last line of a file does not end in a newline character; lines are returned including the trailing
newline when it is present.

The following function is the primary interface of this module:

```
input ([files[, inplace[, backup ]]])
Create an instance of the FileInput class. The instance will be used as global state for the functions of this
module, and is also returned to use during iteration.
```

The following functions use the global state created by input(); if there is no active state, RuntimeError is
raised.

filename ()
Return the name of the file currently being read. Before the first line has been read, returns None.

lineno ()
Return the cumulative line number of the line that has just been read. Before the first line has been read, returns
0. After the last line of the last file has been read, returns the line number of that line.

filelineno ()
Return the line number in the current file. Before the first line has been read, returns 0. After the last line of the last file has been read, returns the line number of that line within the file.

**isfirstline()**
Returns true the line just read is the first line of its file, otherwise returns false.

**isstdin()**
Returns true if the last line was read from sys.stdin, otherwise returns false.

**nextfile()**
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

**close()**
Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

```python
class FileInput([files, inplace[, backup ]])
```

Class `FileInput` is the implementation; its methods `filename()`, `lineno()`, `fileline()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it has a `readline()` method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

Optional in-place filtering: if the keyword argument `inplace=1` is passed to `input()` or to the `FileInput` constructor, the file is moved to a backup file and standard output is directed to the input file. This makes it possible to write a filter that rewrites its input file in place. If the keyword argument `backup='.<some extension>'` is also given, it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `' .bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

**Caveat:** The current implementation does not work for MS-DOS 8+3 filesystems.

## 5.11 xreadlines — Efficient iteration over a file

New in version 2.1.

This module defines a new object type which can efficiently iterate over the lines of a file. An xreadlines object is a sequence type which implements simple in-order indexing beginning at 0, as required by `for` statement or the `filter()` function.

Thus, the code

```python
import xreadlines, sys

for line in xreadlines.xreadlines(sys.stdin):
    pass
```

has approximately the same speed and memory consumption as
while 1:
    lines = sys.stdin.readlines(8*1024)
    if not lines: break
    for line in lines:
        pass

except the clarity of the for statement is retained in the former case.

**xreadlines** *(fileobj)*
Return a new xreadlines object which will iterate over the contents of fileobj. fileobj must have a readlines() method that supports the sizehint parameter.

An xreadlines object s supports the following sequence operation:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[i]</td>
<td>i'th line of s</td>
</tr>
</tbody>
</table>

If successive values of i are not sequential starting from 0, this code will raise RuntimeError.

After the last line of the file is read, this code will raise an IndexError.

5.12 **calendar** — General calendar-related functions

This module allows you to output calendars like the UNIX cal program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use setfirstweekday() to set the first day of the week to Sunday (6) or to any other weekday.

**setfirstweekday** *(weekday)*
Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience. For example, to set the first weekday to Sunday:

```python
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

**firstweekday** ()
Returns the current setting for the weekday to start each week.

**isleap** *(year)*
Returns true if year is a leap year.

**leapdays** *(y1, y2)*
Returns the number of leap years in the range [y1...y2).

**weekday** *(year, month, day)*
Returns the day of the week (0 is Monday) for year (1970...), month (1-12), day (1-31).

**monthrange** *(year, month)*
Returns weekday of first day of the month and number of days in month, for the specified year and month.

**monthcalendar** *(year, month)*
Returns a matrix representing a month’s calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by setfirstweekday().
**prmonth**(theyear, themonth[, w[, l]])
Prints a month’s calendar as returned by month().

**month**(theyear, themonth[, w[, l]])
Returns a month’s calendar in a multi-line string. If w is provided, it specifies the width of the date columns, which are centered. If l is given, it specifies the number of lines that each week will use. Depends on the first weekday as set by setfirstweekday().

**pcreal**(year[, w[, l[, c]]])
Prints the calendar for an entire year as returned by calendar().

**calendar**(year[, w[, l[, c]]])
Returns a 3-column calendar for an entire year as a multi-line string. Optional parameters w, l, and c are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as set by setfirstweekday().

**timegm**(tuple)
An unrelated but handy function that takes a time tuple such as returned by the gmtime() function in the time module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, time.gmtime() and timegm() are each others’ inverse.

See Also:
Module time (section 6.9):
Low-level time related functions.

5.13 cmd — Support for line-oriented command interpreters

The Cmd class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

class Cmd()
A Cmd instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate Cmd itself; rather, it’s useful as a superclass of an interpreter class you define yourself in order to inherit Cmd’s methods and encapsulate action methods.

5.13.1 Cmd Objects

A Cmd instance has the following methods:

**cmdloop**(intro)
Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the intro class member).

If the readline module is loaded, input will automatically inherit bash-like history-list editing (e.g. Ctrl-P scrolls back to the last command, Ctrl-N forward to the next one, Ctrl-F moves the cursor to the right non-destructively, Ctrl-B moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string ‘EOF’.

An interpreter instance will recognize a command name ‘foo’ if and only if it has a method do_foo(). As a special case, a line beginning with the character ‘?’ is dispatched to the method do_help(). As another special case, a line beginning with the character ‘!’ is dispatched to the method do_shell (if such a method is defined).
All subclasses of `Cmd` inherit a predefined `do_help`. This method, called with an argument `bar`, invokes the corresponding method `help_bar()`. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*()` methods), and also lists any undocumented commands.

`onecmd (str)`
Interpret the argument as though it had been typed in in response to the prompt.

`emptyline()`
Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

`default (line)`
Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

`precmd()`
Hook method executed just before the input prompt is issued. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

`postcmd()`
Hook method executed just after a command dispatch is finished. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

`preloop()`
Hook method executed once when `cmdloop()` is called. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

`postloop()`
Hook method executed once when `cmdloop()` is about to return. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

Instances of `Cmd` subclasses have some public instance variables:

`prompt`
The prompt issued to solicit input.

`identchars`
The string of characters accepted for the command prefix.

`lastcmd`
The last nonempty command prefix seen.

`intro`
A string to issue as an intro or banner. May be overridden by giving the `cmdloop()` method an argument.

`doc_header`
The header to issue if the help output has a section for documented commands.

`misc_header`
The header to issue if the help output has a section for miscellaneous help topics (that is, there are `help_*()` methods without corresponding `do_*()` methods).

`undoc_header`
The header to issue if the help output has a section for undocumented commands (that is, there are `do_*()` methods without corresponding `help_*()` methods).

`ruler`
The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to `=`.

`use_rawinput`
A flag, defaulting to true. If true, `cmdloop()` uses `raw_input()` to display a prompt and read the next command; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used.
5.14  `shlex` — Simple lexical analysis

New in version 1.5.2.

The `shlex` class makes it easy to write lexical analyzers for simple syntaxes resembling that of the UNIX shell. This will often be useful for writing minilanguages, e.g. in run control files for Python applications.

```python
class shlex([stream[, file]])
```

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file- or stream-like object with `read()` and `readline()` methods. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` member. If the stream argument is omitted or equal to `sys.stdin`, this second argument defaults to “stdin”.

See Also:
`Module ConfigParser` (section 5.9):
Parser for configuration files similar to the Windows ‘.ini’ files.

5.14.1  `shlex` Objects

A `shlex` instance has the following methods:

```python
get_token()  
```

Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, an empty string is returned.

```python
push_token(str)
```

Push the argument onto the token stack.

```python
read_token()
```

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

```python
sourcehook(filename)
```

When `shlex` detects a source request (see `source` below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (e.g. `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding ‘close’ hook, but a `shlex` instance will call the `close()` method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the `push_source()` and `pop_source()` methods.

```python
push_source(stream[, filename])
```

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the `sourcehook` method. New in version 2.1.

```python
pop_source()
```

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer
reaches EOF on a stacked input stream. New in version 2.1.

**error_leader** ([`file`, `line`])

This method generates an error message leader in the format of a UNIX C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage `shlex` users to generate error messages in the standard, parseable format understood by Emacs and other UNIX tools.

Instances of `shlex` subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

**commenters**

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

**wordchars**

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore.

**whitespace**

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

**quotes**

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

**infile**

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

**instream**

The input stream from which this `shlex` instance is reading characters.

**source**

This member is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will opened as a filename and input taken from that stream until EOF, at which point the `close()` method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

**debug**

If this member is numeric and 1 or more, a `shlex` instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

Note that any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token.

Quote and comment characters are not recognized within words. Thus, the bare words `ain’t` and `ain#t` would be returned as single tokens by the default parser.

**lineno**

Source line number (count of newlines seen so far plus one).

**token**

The token buffer. It may be useful to examine this when catching exceptions.
CHAPTER SIX

Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the UNIX or C interfaces, but they are available on most other systems as well. Here’s an overview:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>os</td>
<td>Miscellaneous OS interfaces.</td>
</tr>
<tr>
<td>os.path</td>
<td>Common pathname manipulations.</td>
</tr>
<tr>
<td>dircache</td>
<td>Return directory listing, with cache mechanism.</td>
</tr>
<tr>
<td>stat</td>
<td>Utilities for interpreting the results of os.stat(), os.lstat() and os.fstat().</td>
</tr>
<tr>
<td>statcache</td>
<td>Stat files, and remember results.</td>
</tr>
<tr>
<td>statvfs</td>
<td>Constants for interpreting the result of os.statvfs().</td>
</tr>
<tr>
<td>filecmp</td>
<td>Compare files efficiently.</td>
</tr>
<tr>
<td>time</td>
<td>Time access and conversions.</td>
</tr>
<tr>
<td>sched</td>
<td>General purpose event scheduler.</td>
</tr>
<tr>
<td>mutex</td>
<td>Lock and queue for mutual exclusion.</td>
</tr>
<tr>
<td>getpass</td>
<td>Portable reading of passwords and retrieval of the userid.</td>
</tr>
<tr>
<td>curses</td>
<td>An interface to the curses library, providing portable terminal handling.</td>
</tr>
<tr>
<td>curses.textpad</td>
<td>Emacs-like input editing in a curses window.</td>
</tr>
<tr>
<td>curses.wrapper</td>
<td>Terminal configuration wrapper for curses programs.</td>
</tr>
<tr>
<td>curses.ascii</td>
<td>Constants and set-membership functions for ASCII characters.</td>
</tr>
<tr>
<td>curses.panel</td>
<td>A panel stack extension that adds depth to curses windows.</td>
</tr>
<tr>
<td>getopt</td>
<td>Portable parser for command line options; support both short and long option names.</td>
</tr>
<tr>
<td>tempfile</td>
<td>Generate temporary file names.</td>
</tr>
<tr>
<td>errno</td>
<td>Standard errno system symbols.</td>
</tr>
<tr>
<td>glob</td>
<td>UNIX shell style pathname pattern expansion.</td>
</tr>
<tr>
<td>fnmatch</td>
<td>UNIX shell style filename pattern matching.</td>
</tr>
<tr>
<td>shutil</td>
<td>High-level file operations, including copying.</td>
</tr>
<tr>
<td>locale</td>
<td>Internationalization services.</td>
</tr>
<tr>
<td>gettext</td>
<td>Multilingual internationalization services.</td>
</tr>
</tbody>
</table>

6.1 os — Miscellaneous OS interfaces

This module provides a more portable way of using operating system (OS) dependent functionality than importing an OS dependent built-in module like posix or nt.

This module searches for an OS dependent built-in module like mac or posix and exports the same functions and data as found there. The design of all Python’s built-in OS dependent modules is such that as long as the same functionality is available, it uses the same interface; e.g., the function os.stat(path) returns stat information about path in the same format (which happens to have originated with the POSIX interface).
Extensions peculiar to a particular OS are also available through the `os` module, but using them is of course a threat to portability!

Note that after the first time `os` is imported, there is no performance penalty in using functions from `os` instead of directly from the OS dependent built-in module, so there should be no reason not to use `os`!

**exception error**

This exception is raised when a function returns a system-related error (e.g., not for illegal argument types). This is also known as the built-in exception `OSError`. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

See the module `errno`, which contains names for the error codes defined by the underlying operating system.

When exceptions are classes, this exception carries two attributes, `errno` and `strerror`. The first holds the value of the `errno` variable, and the latter holds the corresponding error message from `strerror()`. For exceptions that involve a file system path (e.g. `chdir()` or `unlink()`), the exception instance will contain a third attribute, `filename`, which is the file name passed to the function.

When exceptions are strings, the string for the exception is ‘`OSError`’.

**name**

The name of the OS dependent module imported. The following names have currently been registered: ‘`posix`’, ‘`nt`’, ‘`dos`’, ‘`mac`’, ‘`os2`’, ‘`ce`’, ‘`java`’.

**path**

The corresponding OS dependent standard module for pathname operations, e.g., `posixpath` or `macpath`. Thus, given the proper imports, `os.path.split(file)` is equivalent to but more portable than `posixpath.split(file)`. Note that this is also a valid module: it may be imported directly as `os.path`.

### 6.1.1 Process Parameters

These functions and data items provide information and operate on the current process and user.

**environ**

A mapping object representing the string environment. For example, `environ[‘HOME’]` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

If the platform supports the `putenv()` function, this mapping may be used to modify the environment as well as query the environment. `putenv()` will be called automatically when the mapping is modified.

If `putenv()` is not provided, this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

**chdir**(path)

These functions are described in “Files and Directories” (section 6.1.4).

**ctermid()**

Return the filename corresponding to the controlling terminal of the process. Availability: UNIX.

**getegid()**

Return the current process’ effective group id. Availability: UNIX.

**geteuid()**

Return the current process’ effective user id. Availability: UNIX.

**getgid()**

Return the current process’ group id. Availability: UNIX.

**getgroups()**

Return list of supplemental group ids associated with the current process. Availability: UNIX.

**getlogin()**
Return the actual login name for the current process, even if there are multiple login names which map to the same user id. Availability: UNIX.

getpgrp()
Return the current process group id. Availability: UNIX.

g getpid()
Return the current process id. Availability: UNIX, Windows.

getppid()
Return the parent’s process id. Availability: UNIX.

g etuid()
Return the current process’ user id. Availability: UNIX.

putenv (varname, value)
Set the environment variable named varname to the string value. Such changes to the environment affect sub-processes started with os.system(), popen() or fork() and execv(). Availability: most flavors of UNIX, Windows.

When putenv() is supported, assignments to items in os.environ are automatically translated into corresponding calls to putenv(); however, calls to putenv() don’t update os.environ, so it is actually preferable to assign to items of os.environ.

setegid (egid)
Set the current process’s effective group id. Availability: UNIX.

seteuid (euid)
Set the current process’s effective user id. Availability: UNIX.

setgid (gid)
Set the current process’ group id. Availability: UNIX.

setpgrp()
Calls the system call setpgrp() or setpgrp(0, 0) depending on which version is implemented (if any). See the UNIX manual for the semantics. Availability: UNIX.

setpgid (pid, pgrp)
Calls the system call setpgid(). See the UNIX manual for the semantics. Availability: UNIX.

setreuid (ruid, euid)
Set the current process’s real and effective user ids. Availability: UNIX.

setregid (rgid, egid)
Set the current process’s real and effective group ids. Availability: UNIX.

setsid()
Calls the system call setsid(). See the UNIX manual for the semantics. Availability: UNIX.

setuid (uid)
Set the current process’ user id. Availability: UNIX.

strerror (code)
Return the error message corresponding to the error code in code. Availability: UNIX, Windows.

umask (mask)
Set the current numeric umask and returns the previous umask. Availability: UNIX, Windows.

uname()
Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: (sysname, nodename, release, version, machine). Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is socket.gethostname() or even socket.gethostbyaddr(socket.gethostname()). Availability: recent flavors of UNIX.
6.1.2 File Object Creation

These functions create new file objects.

`fdopen(fd[, mode[, bufsize]])`
Return an open file object connected to the file descriptor `fd`. The `mode` and `bufsize` arguments have the same meaning as the corresponding arguments to the built-in `open()` function. Availability: Macintosh, UNIX, Windows.

`popen(command[, mode[, bufsize]])`
Open a pipe to or from `command`. The return value is an open file object connected to the pipe, which can be read or written depending on whether `mode` is ’r’ (default) or ’w’. The `bufsize` argument has the same meaning as the corresponding argument to the built-in `open()` function. The exit status of the command (encoded in the format specified for `wait()` is available as the return value of the `close()` method of the file object, except that when the exit status is zero (termination without errors), `None` is returned. Availability: UNIX, Windows.

Changed in version 2.0: This function worked unreliably under Windows in earlier versions of Python. This was due to the use of the `_popen()` function from the libraries provided with Windows. Newer versions of Python do not use the broken implementation from the Windows libraries.

`tmpfile()`
Return a new file object opened in update mode (’w+’). The file has no directory entries associated with it and will be automatically deleted once there are no file descriptors for the file. Availability: UNIX.

For each of these `popen()` variants, if `bufsize` is specified, it specifies the buffer size for the I/O pipes. `mode`, if provided, should be the string ’b’ or ’t’; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for `mode` is ’t’.

`popen2(cmd[, bufsize[, mode]])`
Executes `cmd` as a sub-process. Returns the file objects `(child_stdin, child_stdout)`. New in version 2.0.

`popen3(cmd[, bufsize[, mode]])`
Executes `cmd` as a sub-process. Returns the file objects `(child_stdin, child_stdout, child_stderr)`. New in version 2.0.

`popen4(cmd[, bufsize[, mode]])`
Executes `cmd` as a sub-process. Returns the file objects `(child_stdin, child_stdout_and_stderr)`. New in version 2.0.

This functionality is also available in the `popen2` module using functions of the same names, but the return values of those functions have a different order.

6.1.3 File Descriptor Operations

These functions operate on I/O streams referred to using file descriptors.

`close(fd)`

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by `open()` or `pipe()`. To close a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

`dup(fd)`
Return a duplicate of file descriptor `fd`. Availability: Macintosh, UNIX, Windows.

`dup2(fd, fd2)`
Duplicate file descriptor `fd` to `fd2`, closing the latter first if necessary. Availability: UNIX, Windows.

`fpathconf(fd, name)`
Return system configuration information relevant to an open file. `name` specifies the configuration value to
retrieve; it may be a string which is the name of a defined system value; these names are specified in a number
of standards (POSIX.1, Unix95, Unix98, and others). Some platforms define additional names as well. The
names known to the host operating system are given in the pathconf_names dictionary. For configuration
variables not included in that mapping, passing an integer for name is also accepted. Availability: UNIX.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by
the host system, even if it is included in pathconf_names, an OSError is raised with errno.EINVAL for
the error number.

fstat (fd)
Return status for file descriptor fd, like stat(). Availability: UNIX, Windows.

fstatvfs (fd)
Return information about the filesystem containing the file associated with file descriptor fd, like statvfs().
Availability: UNIX.

ftruncate (fd, length)
Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size. Availability: UNIX.

isatty (fd)
Return 1 if the file descriptor fd is open and connected to a tty(-like) device, else 0. Availability: UNIX.

lseek (fd, pos, how)
Set the current position of file descriptor fd to position pos, modified by how: 0 to set the position relative to
the beginning of the file; 1 to set it relative to the current position; 2 to set it relative to the end of the file.
Availability: Macintosh, UNIX, Windows.

open (file, flags, mode)
Open the file file and set various flags according to flags and possibly its mode according to mode. The default
mode is 0777 (octal), and the current umask value is first masked out. Return the file descriptor for the newly

For a description of the flag and mode values, see the C run-time documentation; flag constants (like O_RDONLY
and O_WRONLY) are defined in this module too (see below).

Note: this function is intended for low-level I/O. For normal usage, use the built-in function open(), which
returns a “file object” with read() and write() methods (and many more).

openpty ()
Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty,
respectively. For a (slightly) more portable approach, use the pty module. Availability: Some flavors of UNIX

pipe ()
Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively. Availability:
UNIX, Windows.

read (fd, n)
Read at most n bytes from file descriptor fd. Return a string containing the bytes read. Availability: Macintosh,
UNIX, Windows.

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by open()
or pipe(). To read a “file object” returned by the built-in function open() or by popen() or fdopen(), or
sys.stdin, use its read() or readline() methods.

tcgetpgrp (fd)
Return the process group associated with the terminal given by fd (an open file descriptor as returned by
open()). Availability: UNIX.

tcsetpgrp (fd, pg)
Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg. Availability: UNIX.

ttyname (fd)
Return a string which specifies the terminal device associated with file-descriptor \texttt{fd}. If \texttt{fd} is not associated with a terminal device, an exception is raised. Availability: UNIX.

\texttt{write} (\texttt{fd, str})

Write the string \texttt{str} to file descriptor \texttt{fd}. Return the number of bytes actually written. Availability: Macintosh, UNIX, Windows.

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by \texttt{open()} or \texttt{pipe()}. To write a “file object” returned by the built-in function \texttt{open()} or by \texttt{popen()} or \texttt{fdopen()}, or \texttt{sys.stdout} or \texttt{sys.stderr}, use its \texttt{write()} method.

The following data items are available for use in constructing the \texttt{flags} parameter to the \texttt{open()} function.

\begin{itemize}
  \item \texttt{O_RDONLY}
  \item \texttt{O_WRONLY}
  \item \texttt{O_RDWR}
  \item \texttt{O_NONBLOCK}
  \item \texttt{O_APPEND}
  \item \texttt{O_DSYNC}
  \item \texttt{O_SYNC}
  \item \texttt{O_NOCTTY}
  \item \texttt{O_CREAT}
  \item \texttt{O_TRUNC}
\end{itemize}

Options for the \texttt{flag} argument to the \texttt{open()} function. These can be bit-wise OR’d together. Availability: Macintosh, UNIX, Windows.

\begin{itemize}
  \item \texttt{O_BINARY}
\end{itemize}

Option for the \texttt{flag} argument to the \texttt{open()} function. This can be bit-wise OR’d together with those listed above. Availability: Macintosh, Windows.

### 6.1.4 Files and Directories

\texttt{access} (\texttt{path, mode})

Check read/write/execute permissions for this process or existence of file \texttt{path}. \texttt{mode} should be \texttt{F_OK} to test the existence of \texttt{path}, or it can be the inclusive OR of one or more of \texttt{R_OK}, \texttt{W_OK}, and \texttt{X_OK} to test permissions. Return 1 if access is allowed, 0 if not. See the UNIX man page \texttt{access(2)} for more information. Availability: UNIX, Windows.

\begin{itemize}
  \item \texttt{F_OK}
    \texttt{Value to pass as the \texttt{mode} parameter of \texttt{access()} to test the existence of \texttt{path}.}
  \item \texttt{R_OK}
    \texttt{Value to include in the \texttt{mode} parameter of \texttt{access()} to test the readability of \texttt{path}.}
  \item \texttt{W_OK}
    \texttt{Value to include in the \texttt{mode} parameter of \texttt{access()} to test the writability of \texttt{path}.}
  \item \texttt{X_OK}
    \texttt{Value to include in the \texttt{mode} parameter of \texttt{access()} to determine if \texttt{path} can be executed.}
\end{itemize}

\texttt{chdir} (\texttt{path})

Change the current working directory to \texttt{path}. Availability: Macintosh, UNIX, Windows.

\texttt{getcwd}()

Return a string representing the current working directory. Availability: Macintosh, UNIX, Windows.

\texttt{chmod} (\texttt{path, mode})
Change the mode of `path` to the numeric `mode`. Availability: UNIX, Windows.

**chown** *(path, uid, gid)*

Change the owner and group id of `path` to the numeric `uid` and `gid`. Availability: UNIX.

**link** *(src, dst)*

Create a hard link pointing to `src` named `dst`. Availability: UNIX.

**listdir** *(path)*

Return a list containing the names of the entries in the directory. The list is in arbitrary order. It does not include the special entries `. ` and `..` even if they are present in the directory. Availability: Macintosh, UNIX, Windows.

**lstat** *(path)*

Like `stat()`, but do not follow symbolic links. Availability: UNIX.

**mkdir** *(path[, mode])*  
Create a directory named `path` with numeric mode `mode`. The default `mode` is 0777 (octal). On some systems, `mode` is ignored. Where it is used, the current umask value is first masked out. Availability: Macintosh, UNIX, Windows.

**mkfifo** *(path[, mode])*  
Create a FIFO (a named pipe) named `path` with numeric mode `mode`. The default `mode` is 0666 (octal). The current umask value is first masked out from the mode. Availability: UNIX.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between “client” and “server” type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn’t open the FIFO — it just creates the rendezvous point.

**makedirs** *(path[, mode])*  
Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory. Throws an `error` exception if the leaf directory already exists or cannot be created. The default `mode` is 0777 (octal). New in version 1.5.2.

**pathconf** *(path, name)*

Return system configuration information relevant to a named file. `name` specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix95, Unix98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for `name` is also accepted. Availability: UNIX.

If `name` is a string and is not known, `ValueError` is raised. If a specific value for `name` is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

**pathconf_names**

Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: UNIX.

**readlink** *(path)*

Return a string representing the path to which the symbolic link points. Availability: UNIX.

**remove** *(path)*

Remove the file `path`. See `rmdir()` below to remove a directory. This is identical to the `unlink()` function documented below. Availability: Macintosh, UNIX, Windows.

**removedirs** *(path)*

Recursive directory removal function. Works like `rmdir()` except that, if the leaf directory is successfully removed, directories corresponding to rightmost path segments will be pruned way until either the whole path is consumed or an error is raised (which is ignored, because it generally means that a parent directory is not
empty). Throws an error exception if the leaf directory could not be successfully removed. New in version 1.5.2.

rename (src, dst)
Rename the file or directory src to dst. Availability: Macintosh, UNIX, Windows.

renames (old, new)
Recursive directory or file renaming function. Works like rename(), except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using removedirs().

Note: this function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file. New in version 1.5.2.

rmdir (path)
Remove the directory path. Availability: Macintosh, UNIX, Windows.

stat (path)
Perform a stat() system call on the given path. The return value is a tuple of at least 10 integers giving the most important (and portable) members of the stat structure, in the order st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime. More items may be added at the end by some implementations. Note that on the Macintosh, the time values are floating point values, like all time values on the Macintosh. (On MS Windows, some items are filled with dummy values.) Availability: Macintosh, UNIX, Windows.

Note: The standard module stat defines functions and constants that are useful for extracting information from a stat structure.

statvfs (path)
Perform a statvfs() system call on the given path. The return value is a tuple of 10 integers giving the most common members of the statvfs structure, in the order f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_name. Availability: UNIX.

Note: The standard module statvfs defines constants that are useful for extracting information from a statvfs structure.

symlink (src, dst)
Create a symbolic link pointing to src named dst. Availability: UNIX.

tempnam ([dir[, prefix]])
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in the directory dir or a common location for temporary files if dir is omitted or None. If given and not None, prefix is used to provide a short prefix to the filename. Applications are responsible for properly creating and managing files created using paths returned by tempnam(); no automatic cleanup is provided.

tmpnam ()
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in a common location for temporary files. Applications are responsible for properly creating and managing files created using paths returned by tmpnam(); no automatic cleanup is provided.

TMP_MAX
The maximum number of unique names that tmpnam() will generate before reusing names.

unlink (path)
Remove the file path. This is the same function as remove(); the unlink() name is its traditional UNIX name. Availability: Macintosh, UNIX, Windows.

utime (path, times)
Set the access and modified times of the file specified by path. If times is None, then the file’s access and modified times are set to the current time. Otherwise, times must be a 2-tuple of numbers, of the form (atime,
which is used to set the access and modified times, respectively. Changed in version 2.0: Added support for None for times. Availability: Macintosh, UNIX, Windows.

6.1.5 Process Management

These functions may be used to create and manage processes.

The various exec*() functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the argv[0] passed to a program’s main(). For example, ‘os.execv('/bin/echo', ['foo', 'bar'])’ will only print ‘bar’ on standard output; ‘foo’ will seem to be ignored.

abort()
Generate a SIGABRT signal to the current process. On UNIX, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of 3. Be aware that programs which use signal.signal() to register a handler for SIGABRT will behave differently. Availability: UNIX, Windows.

exec1(path, arg0, arg1, ...)
This is equivalent to ‘execv(path, (arg0, arg1, ...))’. Availability: UNIX, Windows.

execle(path, arg0, arg1, ..., env)
This is equivalent to ‘execve(path, (arg0, arg1, ...), env)’. Availability: UNIX, Windows.

execlp(path, arg0, arg1, ...)
This is equivalent to ‘execvp(path, (arg0, arg1, ...))’. Availability: UNIX, Windows.

execv(path, args)
Execute the executable path with argument list args, replacing the current process (i.e., the Python interpreter). The argument list may be a tuple or list of strings. Availability: UNIX, Windows.

execve(path, args, env)
Execute the executable path with argument list args, and environment env, replacing the current process (i.e., the Python interpreter). The argument list may be a tuple or list of strings. The environment must be a dictionary mapping strings to strings. Availability: UNIX, Windows.

execvp(path, args)
This is like ‘execv(path, args)’ but duplicates the shell’s actions in searching for an executable file in a list of directories. The directory list is obtained from environ['PATH']. Availability: UNIX, Windows.

execvpe(path, args, env)
This is a cross between execve() and execvp(). The directory list is obtained from env['PATH']. Availability: UNIX, Windows.

_exit(n)
Exit to the system with status n, without calling cleanup handlers, flushing stdio buffers, etc. Availability: UNIX, Windows.

Note: the standard way to exit is sys.exit(n). _exit() should normally only be used in the child process after a fork().

fork()
Fork a child process. Return 0 in the child, the child’s process id in the parent. Availability: UNIX.

forkpty()
Fork a child process, using a new pseudo-terminal as the child’s controlling terminal. Return a pair of (pid, fd), where pid is 0 in the child, the new child’s process id in the parent, and fd is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the pty module. Availability: Some flavors of UNIX.

kill(pid, sig)
Kill the process *pid* with signal *sig*. Availability: UNIX.

**nice (increment)**

Add increment to the process’s “niceness”. Return the new niceness. Availability: UNIX.

**plock (op)**

Lock program segments into memory. The value of *op* (defined in `<sys/lock.h>`) determines which segments are locked. Availability: UNIX.

**spawnv (mode, path, args)**

Execute the program *path* in a new process, passing the arguments specified in *args* as command-line parameters. *args* may be a list or a tuple. *mode* is a magic operational constant. See the Visual C++ Runtime Library documentation for further information; the constants are exposed to the Python programmer as listed below. Availability: UNIX, Windows. New in version 1.5.2.

**spawnve (mode, path, args, env)**

Execute the program *path* in a new process, passing the arguments specified in *args* as command-line parameters and the contents of the mapping *env* as the environment. *args* may be a list or a tuple. *mode* is a magic operational constant. See the Visual C++ Runtime Library documentation for further information; the constants are exposed to the Python programmer as listed below. Availability: UNIX, Windows. New in version 1.5.2.

**P_WAIT**

**P_NOWAIT**

**P_NOWAITO**

Possible values for the *mode* parameter to *spawnv()* and *spawnve()* . Availability: UNIX, Windows. New in version 1.5.2.

**P_OVERLAY**

**P_DETACH**

Possible values for the *mode* parameter to *spawnv()* and *spawnve()* . These are less portable than those listed above. Availability: Windows. New in version 1.5.2.

**startfile (path)**

Start a file with its associated application. This acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the DOS *start* command: the file is opened with whatever application (if any) its extension is associated.

*startfile()* returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application’s exit status. The *path* parameter is relative to the current directory. If you want to use an absolute path, make sure the first character is not a slash (`/`); the underlying Win32 *ShellExecute()* function doesn’t work it is. Use the *os.path.normpath()* function to ensure that the path is properly encoded for Win32. Availability: Windows. New in version 2.0.

**system (command)**

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function *system()* , and has the same limitations. Changes to *posix.environ*, *sys.stdin*, etc. are not reflected in the environment of the executed command. The return value is the exit status of the process encoded in the format specified for *wait()* , except on Windows 95 and 98, where it is always 0. Note that POSIX does not specify the meaning of the return value of the C *system()* function, so the return value of the Python function is system-dependent. Availability: UNIX, Windows.

**times ()**

Return a 5-tuple of floating point numbers indicating accumulated (CPU or other) times, in seconds. The items are: user time, system time, children’s user time, children’s system time, and elapsed real time since a fixed point in the past, in that order. See the UNIX manual page *times*(2) or the corresponding Windows Platform API documentation. Availability: UNIX, Windows.

**wait ()**

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if
the signal number is zero); the high bit of the low byte is set if a core file was produced. Availability: UNIX.

\texttt{waitpid}(\textit{pid}, \textit{options})

Wait for completion of a child process given by process id \textit{pid}, and return a tuple containing its process id and exit status indication (encoded as for \texttt{wait()}). The semantics of the call are affected by the value of the integer \textit{options}, which should be 0 for normal operation. Availability: UNIX.

If \textit{pid} is greater than 0, \texttt{waitpid()} requests status information for that specific process. If \textit{pid} is 0, the request is for the status of any child in the process group of the current process. If \textit{pid} is 1, the request pertains to any child of the current process. If \textit{pid} is less than 1, status is requested for any process in the process group \texttt{–pid} (the absolute value of \textit{pid}).

\texttt{WNOHANG}

The option for \texttt{waitpid()} to avoid hanging if no child process status is available immediately. Availability: UNIX.

The following functions take a process status code as returned by \texttt{system()}, \texttt{wait()}, or \texttt{waitpid()} as a parameter. They may be used to determine the disposition of a process.

\texttt{WIFSTOPPED}(\textit{status})

Return true if the process has been stopped. Availability: UNIX.

\texttt{WIFSIGNALED}(\textit{status})

Return true if the process exited due to a signal. Availability: UNIX.

\texttt{WIFEXITED}(\textit{status})

Return true if the process exited using the \texttt{exit(2)} system call. Availability: UNIX.

\texttt{WEXITSTATUS}(\textit{status})

If \texttt{WIFEXITED}(\textit{status}) is true, return the integer parameter to the \texttt{exit(2)} system call. Otherwise, the return value is meaningless. Availability: UNIX.

\texttt{WSTOPSIG}(\textit{status})

Return the signal which caused the process to stop. Availability: UNIX.

\texttt{WTERMSIG}(\textit{status})

Return the signal which caused the process to exit. Availability: UNIX.

6.1.6 Miscellaneous System Information

\texttt{confstr}(\textit{name})

Return string-valued system configuration values. \textit{name} specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix95, Unix98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the \texttt{confstr_names} dictionary. For configuration variables not included in that mapping, passing an integer for \textit{name} is also accepted. Availability: UNIX.

If the configuration value specified by \textit{name} isn’t defined, the empty string is returned.

If \textit{name} is a string and is not known, \texttt{ValueError} is raised. If a specific value for \textit{name} is not supported by the host system, even if it is included in \texttt{confstr_names}, an \texttt{OSError} is raised with \texttt{errno.EINVAL} for the error number.

\texttt{confstr_names}

Dictionary mapping names accepted by \texttt{confstr()} to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: UNIX.

\texttt{sysconf}(\textit{name})

Return integer-valued system configuration values. If the configuration value specified by \textit{name} isn’t defined, \texttt{–1} is returned. The comments regarding the \textit{name} parameter for \texttt{confstr()} apply here as well; the dictionary that provides information on the known names is given by \texttt{sysconf_names}. Availability: UNIX.
sysconf_names
Dictionary mapping names accepted by sysconf() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: UNIX.

The follow data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the os.path module.

curdir
The constant string used by the OS to refer to the current directory, e.g. ‘.’ for POSIX or ‘::’ for the Macintosh.

pardir
The constant string used by the OS to refer to the parent directory, e.g. ‘..’ for POSIX or ‘:::’ for the Macintosh.

sep
The character used by the OS to separate pathname components, e.g. ‘/’ for POSIX or ‘:’ for the Macintosh. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use os.path.split() and os.path.join() — but it is occasionally useful.

altsep
An alternative character used by the OS to separate pathname components, or None if only one separator character exists. This is set to ‘/’ on DOS and Windows systems where sep is a backslash.

pathsep
The character conventionally used by the OS to separate search path components (as in PATH), e.g. ‘:’ for POSIX or ‘;’ for DOS and Windows.

defpath
The default search path used by exec*p*() if the environment doesn’t have a ‘PATH’ key.

linesep
The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, e.g. ‘\n’ for POSIX or ‘\r’ for MacOS, or multiple characters, e.g. ‘\r\n’ for MS-DOS and MS Windows.

6.2 os.path — Common pathname manipulations

This module implements some useful functions on pathnames.

abspath(path)
Return a normalized absolutized version of the pathname path. On most platforms, this is equivalent to normpath(join(os.getcwd(), path)). New in version 1.5.2.

basename(path)
Return the base name of pathname path. This is the second half of the pair returned by split(path). Note that the result of this function is different from the UNIX basename program; where basename for ‘/foo/bar/’ returns ‘bar’, the basename() function returns an empty string (‘’).

commonprefix(list)
Return the longest path prefix (taken character-by-character) that is a prefix of all paths in list. If list is empty, return the empty string (‘’). Note that this may return invalid paths because it works a character at a time.

dirname(path)
Return the directory name of pathname path. This is the first half of the pair returned by split(path).

exists(path)
Return true if path refers to an existing path.

expanduser(path)
Return the argument with an initial component of ‘˜’ or ‘˜user’ replaced by that user’s home directory. An initial ‘˜’ is replaced by the environment variable HOME; an initial ‘˜user’ is looked up in the password directory through the built-in module `pwd`. If the expansion fails, or if the path does not begin with a tilde, the path is returned unchanged. On the Macintosh, this always returns path unchanged.

**expandvars**(path)

Return the argument with environment variables expanded. Substrings of the form ‘$name’ or ‘${name}’ are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged. On the Macintosh, this always returns path unchanged.

**getatime**(path)

Return the time of last access of filename. The return value is integer giving the number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible. New in version 1.5.2.

**getmtime**(path)

Return the time of last modification of filename. The return value is integer giving the number of seconds since the epoch (see the time module). Raise os.error if the file does not exist or is inaccessible. New in version 1.5.2.

**getsize**(path)

Return the size, in bytes, of filename. Raise os.error if the file does not exist or is inaccessible. New in version 1.5.2.

**isabs**(path)

Return true if path is an absolute pathname (begins with a slash).

**isfile**(path)

Return true if path is an existing regular file. This follows symbolic links, so both islink() and isfile() can be true for the same path.

**isdir**(path)

Return true if path is an existing directory. This follows symbolic links, so both islink() and isdir() can be true for the same path.

**islink**(path)

Return true if path refers to a directory entry that is a symbolic link. Always false if symbolic links are not supported.

**ismount**(path)

Return true if pathname path is a mount point: a point in a file system where a different file system has been mounted. The function checks whether path’s parent, ‘path/..’, is on a different device than path, or whether ‘path/..’ and path point to the same i-node on the same device — this should detect mount points for all UNIX and POSIX variants.

**join**(path1[, path2[, ...]])

Joins one or more path components intelligently. If any component is an absolute path, all previous components are thrown away, and joining continues. The return value is the concatenation of path1, and optionally path2, etc., with exactly one slash (‘/’) inserted between components, unless path is empty.

**normcase**(path)

Normalize the case of a pathname. On UNIX, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.

**normpath**(path)

Normalize a pathname. This collapses redundant separators and up-level references, e.g. A//B, A//..//B and A/foo/..//B all become A/B. It does not normalize the case (use normcase() for that). On Windows, it converts forward slashes to backward slashes.

**samefile**(path1, path2)

Return true if both pathname arguments refer to the same file or directory (as indicated by device number and
i-node number). Raise an exception if a `os.stat()` call on either pathname fails. Availability: Macintosh, UNIX.

**sameopenfile**(fp1, fp2)
Return true if the file objects fp1 and fp2 refer to the same file. The two file objects may represent different file descriptors. Availability: Macintosh, UNIX.

**samestat**(stat1, stat2)
Return true if the stat tuples stat1 and stat2 refer to the same file. These structures may have been returned by `fstat()`, `lstat()`, or `stat()`. This function implements the underlying comparison used by `samefile()` and `sameopenfile()`. Availability: Macintosh, UNIX.

**split**(path)
Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash; if path ends in a slash, tail will be empty. If there is no slash in path, head will be empty. If path is empty, both head and tail are empty. Trailing slashes are stripped from head unless it is the root (one or more slashes only). In nearly all cases, `join(head, tail)` equals path (the only exception being when there were multiple slashes separating head from tail).

**splitdrive**(path)
Split the pathname path into a pair (drive, tail) where drive is either a drive specification or the empty string. On systems which do not use drive specifications, drive will always be the empty string. In all cases, drive + tail will be the same as path.

**splitext**(path)
Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period.

**walk**(path, visit, arg)
Calls the function visit with arguments (arg, dirname, names) for each directory in the directory tree rooted at path (including path itself, if it is a directory). The argument dirname specifies the visited directory, the argument names lists the files in the directory (gotten from `os.listdir(dirname)`). The visit function may modify names to influence the set of directories visited below dirname, e.g., to avoid visiting certain parts of the tree. (The object referred to by names must be modified in place, using `del` or slice assignment.)

**6.3 dircache — Cached directory listings**

The dircache module defines a function for reading directory listing using a cache, and cache invalidation using the mtime of the directory. Additionally, it defines a function to annotate directories by appending a slash.

The dircache module defines the following functions:

**listdir**(path)
Return a directory listing of path, as gotten from `os.listdir()`. Note that unless path changes, further call to `listdir()` will not re-read the directory structure.

Note that the list returned should be regarded as read-only. (Perhaps a future version should change it to return a tuple?)

**opendir**(path)
Same as `listdir()`. Defined for backwards compatibility.

**annotate**(head, list)
Assume list is a list of paths relative to head, and append, in place, a ‘/’ to each path which points to a directory.

---

152 Chapter 6. Generic Operating System Services
>>> import dircache
>>> a=dircache.listdir('/
>>> a=a[:]
   # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'lib', 'lost+
found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz']
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd/', 'lib/
', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'var/', 'vm
linuz']

6.4 stat — Interpreting stat() results

The stat module defines constants and functions for interpreting the results of os.stat(), os.fstat() and os.lstat() (if they exist). For complete details about the stat(), fstat() and lstat() calls, consult the documentation for your system.

The stat module defines the following functions to test for specific file types:

S_ISDIR(mode)
   Return non-zero if the mode is from a directory.

S_ISCHR(mode)
   Return non-zero if the mode is from a character special device file.

S_ISBLK(mode)
   Return non-zero if the mode is from a block special device file.

S_ISREG(mode)
   Return non-zero if the mode is from a regular file.

S_ISFIFO(mode)
   Return non-zero if the mode is from a FIFO (named pipe).

S_ISLNK(mode)
   Return non-zero if the mode is from a symbolic link.

S_ISSOCK(mode)
   Return non-zero if the mode is from a socket.

Two additional functions are defined for more general manipulation of the file’s mode:

S_IMODE(mode)
   Return the portion of the file’s mode that can be set by os.chmod()—that is, the file’s permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

S_IFMT(mode)
   Return the portion of the file’s mode that describes the file type (used by the S_IS*() functions above).

Normally, you would use the os.path.is*() functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the stat() system call for each test. These are also useful when checking for information about a file that isn’t handled by os.path, like the tests for block and character devices.

All the variables below are simply symbolic indexes into the 10-tuple returned by os.stat(), os.fstat() or os.lstat().
ST_MODE
   Inode protection mode.

ST_INO
   Inode number.

ST_DEV
   Device inode resides on.

ST_NLINK
   Number of links to the inode.

ST_UID
   User id of the owner.

ST_GID
   Group id of the owner.

ST_SIZE
   Size in bytes of a plain file; amount of data waiting on some special files.

ST_ATIME
   Time of last access.

ST_MTIME
   Time of last modification.

ST_CTIME
   Time of last status change (see manual pages for details).

The interpretation of “file size” changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most Unixes (including Linux in particular), the “size” is the number of bytes waiting to be read at the time of the call to os.stat(), os.fstat(), or os.lstat(); this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

Example:
import os, sys
from stat import *

def walktree(dir, callback):
    '''recursively descend the directory rooted at dir,
calling the callback function for each regular file’’’

    for f in os.listdir(dir):
        pathname = '%s/%s' % (dir, f)
        mode = os.stat(pathname)[ST_MODE]
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print 'Skipping %s' % pathname

def visitfile(file):
    print 'visiting', file

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)

6.5 statcache — An optimization of os.stat()

The statcache module provides a simple optimization to os.stat(): remembering the values of previous invocations.

The statcache module defines the following functions:

stat(path)
    This is the main module entry-point. Identical for os.stat(), except for remembering the result for future invocations of the function.

The rest of the functions are used to clear the cache, or parts of it.

reset()
    Clear the cache: forget all results of previous stat() calls.

forget(path)
    Forget the result of stat(path), if any.

forget_prefix(prefix)
    Forget all results of stat(path) for path starting with prefix.

forget_dir(prefix)
    Forget all results of stat(path) for path a file in the directory prefix, including stat(prefix).

forget_except_prefix(prefix)
    Similar to forget_prefix(), but for all path values not starting with prefix.

Example:
6.6  \texttt{statvfs} — Constants used with \texttt{os.statvfs()}

The \texttt{statvfs} module defines constants so interpreting the result if \texttt{os.statvfs()}, which returns a tuple, can be made without remembering “magic numbers.” Each of the constants defined in this module is the \textit{index} of the entry in the tuple returned by \texttt{os.statvfs()} that contains the specified information.

\begin{itemize}
  \item \texttt{F_BSIZE}  
    Preferred file system block size.
  \item \texttt{F_FRSIZE}  
    Fundamental file system block size.
  \item \texttt{F_BLOCKS}  
    Total number of blocks in the filesystem.
  \item \texttt{F_BFREE}  
    Total number of free blocks.
  \item \texttt{F_BAVAIL}  
    Free blocks available to non-super user.
  \item \texttt{F_FILES}  
    Total number of file nodes.
  \item \texttt{F_FFREE}  
    Total number of free file nodes.
  \item \texttt{F_FAVAIL}  
    Free nodes available to non-super user.
  \item \texttt{F_FLAG}  
    Flags. System dependent: see \texttt{statvfs()} man page.
  \item \texttt{F_NAMEMAX}  
    Maximum file name length.
\end{itemize}

6.7  \texttt{filecmp} — File and Directory Comparisons

The \texttt{filecmp} module defines functions to compare files and directories, with various optional time/correctness trade-offs.

The \texttt{filecmp} module defines the following function:

\begin{verbatim}
cmp (f1, f2[, shallow[, use_statcache]])
\end{verbatim}

Compare the files named \texttt{f1} and \texttt{f2}, returning 1 if they seem equal, 0 otherwise.

Unless \texttt{shallow} is given and is false, files with identical \texttt{os.stat()} signatures are taken to be equal. If \texttt{use_statcache} is given and is true, \texttt{statcache.stat()} will be called rather than \texttt{os.stat()}; the default is to use \texttt{os.stat()}.  

156  Chapter 6. Generic Operating System Services
Files that were compared using this function will not be compared again unless their `os.stat()` signature changes. Note that using `use_statcache` true will cause the cache invalidation mechanism to fail — the stale stat value will be used from `statcache`'s cache.

Note that no external programs are called from this function, giving it portability and efficiency.

```python
cmpfiles(dir1, dir2, common[, shallow[, use_statcache ]])
```

Returns three lists of file names: `match`, `mismatch`, `errors`. `match` contains the list of files match in both directories, `mismatch` includes the names of those that don’t, and `errors` lists the names of files which could not be compared. Files may be listed in `errors` because the user may lack permission to read them or many other reasons, but always that the comparison could not be done for some reason.

The `shallow` and `use_statcache` parameters have the same meanings and default values as for `filecmp.cmp()`.

Example:

```python
>>> import filecmp
>>> filecmp.cmp('libundoc.tex', 'libundoc.tex')
1
>>> filecmp.cmp('libundoc.tex', 'lib.tex')
0
```

6.7.1 The `dircmp` class

```python
class dircmp(a, b[, ignore[, hide ]])
```

Construct a new directory comparison object, to compare the directories `a` and `b`. `ignore` is a list of names to ignore, and defaults to `['RCS', 'CVS', 'tags']`. `hide` is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

```python
report()
```

Print (to `sys.stdout`) a comparison between `a` and `b`.

```python
report_partial_closure()
```

Print a comparison between `a` and `b` and common immediate subdirectories.

```python
report_full_closure()
```

Print a comparison between `a` and `b` and common subdirectories (recursively).

```python
left_list
```

Files and subdirectories in `a`, filtered by `hide` and `ignore`.

```python
right_list
```

Files and subdirectories in `b`, filtered by `hide` and `ignore`.

```python
common
```

Files and subdirectories in both `a` and `b`.

```python
left_only
```

Files and subdirectories only in `a`.

```python
right_only
```

Files and subdirectories only in `b`.

```python
common_dirs
```

Subdirectories in both `a` and `b`.

```python
common_files
```

Files in both `a` and `b`.
common funny
Names in both $a$ and $b$, such that the type differs between the directories, or names for which `os.stat()` reports an error.

same files
Files which are identical in both $a$ and $b$.

diff files
Files which are in both $a$ and $b$, whose contents differ.

funny files
Files which are in both $a$ and $b$, but could not be compared.

subdirs
A dictionary mapping names in common dirs to `dir`cmp objects.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

6.8 `popen2` — Subprocesses with accessible I/O streams

This module allows you to spawn processes and connect to their input/output/error pipes and obtain their return codes under UNIX and Windows.

Note that starting with Python 2.0, this functionality is available using functions from the `os` module which have the same names as the factory functions here, but the order of the return values is more intuitive in the `os` module variants.

The primary interface offered by this module is a trio of factory functions. For each of these, if `bufsize` is specified, it specifies the buffer size for the I/O pipes. `mode`, if provided, should be the string `b` or `t`; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for `mode` is `t`.

`popen2(cmd[, bufsize[, mode ]])`
Executes `cmd` as a sub-process. Returns the file objects `(child_stdout, child_stdin)`.

`popen3(cmd[, bufsize[, mode ]])`
Executes `cmd` as a sub-process. Returns the file objects `(child_stdout, child_stdin, child_stderr)`.

`popen4(cmd[, bufsize[, mode ]])`
Executes `cmd` as a sub-process. Returns the file objects `(child_stdout_and_stderr, child_stdin)`.

New in version 2.0.

On UNIX, a class defining the objects returned by the factory functions is also available. These are not used for the Windows implementation, and are not available on that platform.

`class Popen3(cmd[, capturestderr[, bufsize ]])`
This class represents a child process. Normally, `Popen3` instances are created using the `popen2()` and `popen3()` factory functions described above.

If not using one of the helper functions to create `Popen3` objects, the parameter `cmd` is the shell command to execute in a sub-process. The `capturestderr` flag, if true, specifies that the object should capture standard error output of the child process. The default is false. If the `bufsize` parameter is specified, it specifies the size of the I/O buffers to/from the child process.

`class Popen4(cmd[, bufsize ])
Similar to `Popen3`, but always captures standard error into the same file object as standard output. These are typically created using `popen4()`.

New in version 2.0.
6.8.1 Popen3 and Popen4 Objects

Instances of the `Popen3` and `Popen4` classes have the following methods:

`poll()`

Returns -1 if child process hasn’t completed yet, or its return code otherwise.

`wait()`

Waits for and returns the return code of the child process.

The following attributes are also available:

`fromchild`
A file object that provides output from the child process. For `Popen4` instances, this will provide both the standard output and standard error streams.

`tochild`
A file object that provides input to the child process.

`childerr`
Where the standard error from the child process goes is `capturestderr` was true for the constructor, or `None`. This will always be `None` for `Popen4` instances.

`pid`
The process ID of the child process.

6.9 time — Time access and conversions

This module provides various time-related functions. It is always available, but not all functions are available on all platforms.

An explanation of some terminology and conventions is in order.

- The `epoch` is the point where the time starts. On January 1st of that year, at 0 hours, the “time since the epoch” is zero. For UNIX, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.

- The functions in this module do not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for UNIX, it is typically in 2038.

- **Year 2000 (Y2K) issues**: Python depends on the platform’s C library, which generally doesn’t have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Functions accepting a time tuple (see below) generally require a 4-digit year. For backward compatibility, 2-digit years are supported if the module variable `accept2dyear` is a non-zero integer; this variable is initialized to 1 unless the environment variable `PYTHONY2K` is set to a non-empty string, in which case it is initialized to 0. Thus, you can set `PYTHONY2K` to a non-empty string in the environment to require 4-digit years for all year input. When 2-digit years are accepted, they are converted according to the POSIX or X/Open standard: values 69-99 are mapped to 1969-1999, and values 0–68 are mapped to 2000–2068. Values 100–1899 are always illegal. Note that this is new as of Python 1.5.2(a2); earlier versions, up to Python 1.5.1 and 1.5.2a1, would add 1900 to year values below 1900.

- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.

- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
• The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most UNIX systems, the clock “ticks” only 50 or 100 times a second, and on the Mac, times are only accurate to whole seconds.

• On the other hand, the precision of time() and sleep() is better than their UNIX equivalents: times are expressed as floating point numbers, time() returns the most accurate time available (using UNIX gettimeofday() where available), and sleep() will accept a time with a nonzero fraction (UNIX select() is used to implement this, where available).

• The time tuple as returned by gmtime(), localtime(), and strftime(), and accepted by asctime(), mktime() and strftime(), is a tuple of 9 integers:

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>year</td>
<td>(e.g. 1993)</td>
</tr>
<tr>
<td>1</td>
<td>month</td>
<td>range [1,12]</td>
</tr>
<tr>
<td>2</td>
<td>day</td>
<td>range [1,31]</td>
</tr>
<tr>
<td>3</td>
<td>hour</td>
<td>range [0,23]</td>
</tr>
<tr>
<td>4</td>
<td>minute</td>
<td>range [0,59]</td>
</tr>
<tr>
<td>5</td>
<td>second</td>
<td>range [0,61]; see (1) in strftime() description</td>
</tr>
<tr>
<td>6</td>
<td>weekday</td>
<td>range [0,6], Monday is 0</td>
</tr>
<tr>
<td>7</td>
<td>julian day</td>
<td>range [1,366]</td>
</tr>
<tr>
<td>8</td>
<td>daylight savings flag</td>
<td>0, 1 or -1; see below</td>
</tr>
</tbody>
</table>

Note that unlike the C structure, the month value is a range of 1-12, not 0-11. A year value will be handled as described under “Year 2000 (Y2K) issues” above. A -1 argument as daylight savings flag, passed to mktime() will usually result in the correct daylight savings state to be filled in.

The module defines the following functions and data items:

accept2dyear

  Boolean value indicating whether two-digit year values will be accepted. This is true by default, but will be set to false if the environment variable PYTHONY2K has been set to a non-empty string. It may also be modified at run time.

altzone

  The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

asctime([tuple])

  Convert a tuple representing a time as returned by gmtime() or localtime() to a 24-character string of the following form: ’Sun Jun 20 23:21:05 1993’. If tuple is not provided, the current time as returned by localtime() is used. Note: unlike the C function of the same name, there is no trailing newline.

clock()

  Return the current CPU time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of “CPU time”, depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

cftime([secs])

  Convert a time expressed in seconds since the epoch to a string representing local time. If secs is not provided, the current time as returned by time() is used. cftime(secs) is equivalent to asctime(localtime(secs)).

daylight

  Nonzero if a DST timezone is defined.

gmtime([secs])

  Convert a time expressed in seconds since the epoch to a time tuple in UTC in which the dst flag is always zero.
If `secs` is not provided, the current time as returned by `time()` is used. Fractions of a second are ignored. See above for a description of the tuple lay-out.

**localtime([`secs`])**
Like `gmtime()` but converts to local time. The dst flag is set to 1 when DST applies to the given time.

**mktime(tupl)**
This is the inverse function of `localtime()`. Its argument is the full 9-tuple (since the dst flag is needed; use -1 as the dst flag if it is unknown) which expresses the time in local time, not UTC. It returns a floating point number, for compatibility with `time()`. If the input value cannot be represented as a valid time, `OverflowError` is raised.

**sleep(`secs`)**
Suspend execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the `sleep()` following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

**strftime(format[, tuple])**
Convert a tuple representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the `format` argument. If `tuple` is not provided, the current time as returned by `localtime()` is used. `format` must be a string.

The following directives can be embedded in the `format` string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the `strftime()` result:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Locale’s abbreviated weekday name.</td>
<td></td>
</tr>
<tr>
<td>%A</td>
<td>Locale’s full weekday name.</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Locale’s abbreviated month name.</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Locale’s full month name.</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Locale’s appropriate date and time representation.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a decimal number [01,31].</td>
<td></td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a decimal number [00,23].</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a decimal number [001,366].</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a decimal number [00,59].</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td></td>
</tr>
<tr>
<td>%s</td>
<td>Second as a decimal number [00,61].</td>
<td></td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td>(1)</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number [0(Sunday),6].</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td></td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year without century as a decimal number [00,99].</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year with century as a decimal number.</td>
<td></td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name (or by no characters if no time zone exists).</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>A literal ‘%’ character.</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

(1) The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.
Here is an example, a format for dates compatible with that specified in the RFC 822 Internet email standard.  

```python
>>> from time import *
>>> strftime("%a, %d %b %Y %H:%M:%S %Z", localtime())
'Sat, 27 Jan 2001 05:15:05 EST'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C.

On some platforms, an optional field width and precision specification can immediately follow the initial ‘%’ of a directive in the following order; this is also not portable. The field width is normally 2 except for %j where it is 3.

```python
strptime(string[, format])
```
Parse a string representing a time according to a format. The return value is a tuple as returned by gmtime() or localtime(). The format parameter uses the same directives as those used by strftime(): it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime(). The same platform caveats apply; see the local UNIX documentation for restrictions or additional supported directives. If string cannot be parsed according to format, ValueError is raised. Values which are not provided as part of the input string are filled in with default values; the specific values are platform-dependent as the XPG standard does not provide sufficient information to constrain the result.

**Note:** This function relies entirely on the underlying platform’s C library for the date parsing, and some of these libraries are buggy. There’s nothing to be done about this short of a new, portable implementation of strptime().

**Availability:** Most modern UNIX systems.

```python
 time()  
```
Return the time as a floating point number expressed in seconds since the epoch, in UTC. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second.

```python
timezone
```
The offset of the local (non-DST) timezone, in seconds west of UTC (i.e. negative in most of Western Europe, positive in the US, zero in the UK).

```python
tzname
```
A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

**See Also:**
Module locale (section 6.24):
Internationalization services. The locale settings can affect the return values for some of the functions in the time module.

### 6.10 sched — Event scheduler

The sched module defines a class which implements a general purpose event scheduler:

```python
class scheduler(timefunc, delayfunc)
```
The scheduler class defines a generic interface to scheduling events. It needs two functions to actually deal with the “outside world” — timefunc should be callable without arguments, and return a number (the “time”,

---

1The use of %Z is now deprecated, but the %z escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 RFC 822 standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000.
in any units whatsoever). The \texttt{delayfunc} function should be callable with one argument, compatible with the output of \texttt{timefunc}, and should delay that many time units. \texttt{delayfunc} will also be called with the argument \texttt{0} after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Example:

```python
>>> import sched, time
>>> s=sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
... 
>>> def print_some_times():
...     print time.time()
...     s.enter(5, 1, print_time, ())
...     s.enter(10, 1, print_time, ())
...     s.run()
...     print time.time()
... 
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```

### 6.10.1 Scheduler Objects

\texttt{scheduler} instances have the following methods:

- \texttt{enterabs}(\texttt{time, priority, action, argument})
  
  Schedule a new event. The \texttt{time} argument should be a numeric type compatible with the return value of the \texttt{timefunc} function passed to the constructor. Events scheduled for the same \texttt{time} will be executed in the order of their \texttt{priority}.

  Executing the event means executing \texttt{apply(action, argument)}. \texttt{argument} must be a tuple holding the parameters for \texttt{action}.

  Return value is an event which may be used for later cancellation of the event (see \texttt{cancel()}).

- \texttt{enter}(\texttt{delay, priority, action, argument})
  
  Schedule an event for \texttt{delay} more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for \texttt{enterabs}().

- \texttt{cancel}(\texttt{event})
  
  Remove the event from the queue. If \texttt{event} is not an event currently in the queue, this method will raise a \texttt{RuntimeError}.

- \texttt{empty}()
  
  Return true if the event queue is empty.

- \texttt{run}()
  
  Run all scheduled events. This function will wait (using the \texttt{delayfunc} function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

  Either \texttt{action} or \texttt{delayfunc} can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by \texttt{action}, the event will not be attempted in future calls to \texttt{run}().

  If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.
6.11 mutex — Mutual exclusion support

The mutex module defines a class that allows mutual-exclusion via acquiring and releasing locks. It does not require (or imply) threading or multi-tasking, though it could be useful for those purposes.

The mutex module defines the following class:

```
class mutex()
```

Create a new (unlocked) mutex.

A mutex has two pieces of state — a “locked” bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue contains zero or more \((\text{function}, \text{argument})\) pairs representing functions (or methods) waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and its \(\text{function} (\text{argument})\) pair called, implying it now has the lock.

Of course, no multi-threading is implied — hence the funny interface for \text{lock()}\, where a function is called once the lock is acquired.

6.11.1 Mutex Objects

mutex objects have following methods:

```
test()
```

Check whether the mutex is locked.

```
testandset()
```

“Atomic” test-and-set, grab the lock if it is not set, and return true, otherwise, return false.

```
lock(function, argument)
```

Execute \(\text{function}(\text{argument})\), unless the mutex is locked. In the case it is locked, place the function and argument on the queue. See \text{unlock} for explanation of when \(\text{function}(\text{argument})\) is executed in that case.

```
unlock()
```

Unlock the mutex if queue is empty, otherwise execute the first element in the queue.

6.12 getpass — Portable password input

The getpass module provides two functions:

```
getpass([prompt])
```

Prompt the user for a password without echoing. The user is prompted using the string \text{prompt}, which defaults to ‘Password: ’. Availability: Macintosh, UNIX, Windows.

```
getuser()
```

Return the “login name” of the user. Availability: UNIX, Windows.

This function checks the environment variables LOGNAME, USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the \text{pwd} module, otherwise, an exception is raised.

6.13 curses — Terminal handling for character-cell displays

Changed in version 1.6: Added support for the ncurses library and converted to a package.

The curses module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.
While curses is most widely used in the UNIX environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of UNIX.

See Also:

Module `curses.ascii` (section 6.16):
Utilities for working with ASCII characters, regardless of your locale settings.

Module `curses.panel` (section 6.17):
A panel stack extension that adds depth to curses windows.

Module `curses.textpad` (section 6.14):
Editable text widget for curses supporting Emacs-like bindings.

Module `curses.wrapper` (section 6.15):
Convenience function to ensure proper terminal setup and resetting on application entry and exit.

Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond, is available on the Python Web site.

The ‘Demo/curses/’ directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

### 6.13.1 Functions

The module `curses` defines the following exception:

```python
definition
definition
```

---

6.13. `curses` — Terminal handling for character-cell displays

---
Returns the attribute value for displaying text in the specified color. This attribute value can be combined with \texttt{A\_STANDOUT}, \texttt{A\_REVERSE}, and the other \texttt{A\_*} attributes. \texttt{pair\_number()} is the counterpart to this function.

\texttt{curs\_set} (\textit{visibility})
Sets the cursor state. \textit{visibility} can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the “visible” mode is an underline cursor and the “very visible” mode is a block cursor.

\texttt{def\_prog\_mode()}
Saves the current terminal mode as the “program” mode, the mode when the running program is using curses. (Its counterpart is the “shell” mode, for when the program is not in curses.) Subsequent calls to \texttt{reset\_prog\_mode()} will restore this mode.

\texttt{def\_shell\_mode()}
Saves the current terminal mode as the “shell” mode, the mode when the running program is not using curses. (Its counterpart is the “program” mode, when the program is using curses capabilities.) Subsequent calls to \texttt{reset\_shell\_mode()} will restore this mode.

\texttt{delay\_output} (\textit{ms})
Inserts an \textit{ms} millisecond pause in output.

\texttt{doupdate()}
Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The \texttt{doupdate()} ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a \texttt{noutrefresh()} call after write operations such as \texttt{addstr()} have been performed on a window. The normal \texttt{refresh()} call is simply \texttt{noutrefresh()} followed by \texttt{doupdate()}; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing \texttt{noutrefresh()} calls on all windows, followed by a single \texttt{doupdate()}.

\texttt{echo()}
Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

\texttt{endwin()}
De-initialize the library, and return terminal to normal status.

\texttt{erasechar()}
Returns the user’s current erase character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

\texttt{filter()}
The \texttt{filter()} routine, if used, must be called before \texttt{initscr()} is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

\texttt{flash()}
Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as ‘visible bell’ to the audible attention signal produced by \texttt{beep()}.

\texttt{flushinp()}
Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

\texttt{getmouse()}
After \texttt{getch()} returns \texttt{KEY\_MOUSE} to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple \((\textit{id}, \textit{x}, \textit{y}, \textit{z}, \textit{bstate})\). \textit{id} is an ID value used to distinguish multiple devices, and \textit{x}, \textit{y}, \textit{z} are the event’s coordinates. (\textit{z} is currently unused.). \textit{bstate} is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where
n is the button number from 1 to 4: BUTTONn_PRESSED, BUTTONnReleased, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

getsysx()  
Returns the current coordinates of the virtual screen cursor in y and x. If leaveok is currently true, then -1,-1 is returned.

getwin(file)  
Reads window related data stored in the file by an earlier putwin() call. The routine then creates and initializes a new window using that data, returning the new window object.

has_colors()  
Returns true if the terminal can display colors; otherwise, it returns false.

has_ic()  
Returns true if the terminal has insert- and delete- character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

has_il()  
Returns true if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

has_key(ch)  
Takes a key value ch, and returns true if the current terminal type recognizes a key with that value.

halfdelay(tenths)  
Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for tenths tenths of seconds, an exception is raised if nothing has been typed. The value of tenths must be a number between 1 and 255. Use nocbreak() to leave half-delay mode.

init_color(color_number, r, g, b)  
Changes the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of color_number must be between 0 and COLORS. Each of r, g, b, must be a value between 0 and 1000. When init_color() is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if can_change_color() returns 1.

init_pair(pair_number, fg, bg)  
Changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of pair_number must be between 1 and COLOR_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of fg and bg arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

initscr()  
Initialize the library. Returns a WindowObject which represents the whole screen.

isendwin()  
Returns true if endwin() has been called (that is, the curses library has been deinitialized).

keyname(k)  
Return the name of the key numbered k. The name of a key generating printable ASCII character is the key’s character. The name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128-255) is a string consisting of the prefix ‘M-’ followed by the name of the corresponding ASCII character.

killchar()  
Returns the user’s current line kill character. Under Unix operating systems this is a property of the controlling
tty of the curses program, and is not set by the curses library itself.

`longname()`
Returns a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()`.

`meta(yes)`
If `yes` is 1, allow 8-bit characters to be input. If `yes` is 0, allow only 7-bit chars.

`mouseinterval(interval)`
Sets the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and returns the previous interval value. The default value is 200 msec, or one fifth of a second.

`mousemove(mousemask)`
Sets the mouse events to be reported, and returns a tuple `(availmask, oldmask)`. `availmask` indicates which of the specified mouse events can be reported; on complete failure it returns 0. `oldmask` is the previous value of the given window’s mouse event mask. If this function is never called, no mouse events are ever reported.

`napms(ms)`
Sleep for `ms` milliseconds.

`newpad(nlines, ncols)`
Creates and returns a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g., from scrolling or echoing of input) do not occur. The `refresh()` and `noutrefresh()` methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are `pminrow`, `pmincol`, `pminrow`, `pmincol`, `smaxrow`, `smaxcol`; the `p` arguments refer to the upper left corner of the the pad region to be displayed and the `s` arguments define a clipping box on the screen within which the pad region is to be displayed.

`newwin([nlines, ncols,] begin_y, begin_x)`
Return a new window, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is `nlines/ncols`.

By default, the window will extend from the specified position to the lower right corner of the screen.

`nl()`
Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

`nocbreak()`
Leave cbreak mode. Return to normal “cooked” mode with line buffering.

`noecho()`
Leave echo mode. Echoing of input characters is turned off.

`nonl()`
Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

`noqiflush()`
When the noqiflush routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.
noraw()
Leave raw mode. Return to normal “cooked” mode with line buffering.

pair_content (pair_number)
Returns a tuple (fg,bg) containing the colors for the requested color pair. The value of pair_number must be between 0 and COLOR_PAIRS-1.

pair_number (attr)
Returns the number of the color-pair set by the attribute value attr. color_pair() is the counterpart to this function.

putp (string)
Equivalent to puts(str, 1, putchar); emits the value of a specified terminfo capability for the current terminal. Note that the output of putp always goes to standard output.

qiflush ([flag])
If flag is false, the effect is the same as calling noqiflush(). If flag is true, or no argument is provided, the queues will be flushed when these control characters are read.

raw()
Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

reset_prog_mode()
Restores the terminal to “program” mode, as previously saved by def_prog_mode().

reset_shell_mode()
Restores the terminal to “shell” mode, as previously saved by def_shell_mode().

setsyx (y, x)
Sets the virtual screen cursor to y, x. If y and x are both -1, then leaveok is set.

setupterm ([termstr, fd])
Initializes the terminal. termstr is a string giving the terminal name; if omitted, the value of the TERM environment variable will be used. fd is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for sys.stdout will be used.

start_color()
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after initscr().

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the curses module, COLORS and COLOR_PAIRS, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

termattrs()
Returns a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

termsname()
Returns the value of the environment variable TERM, truncated to 14 characters.

tigetflag (capname)
Returns the value of the Boolean capability corresponding to the terminfo capability name capname. The value -1 is returned if capname is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

tigetnum (capname)
Returns the value of the numeric capability corresponding to the terminfo capability name capname. The value -2 is returned if capname is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

tigetstr (capname)
Returns the value of the string capability corresponding to the terminfo capability name \textit{capname}. None is returned if \textit{capname} is not a string capability, or is canceled or absent from the terminal description.

\textbf{tparm} \((\texttt{str[...]}))

Instantiates the string \texttt{str} with the supplied parameters, where \texttt{str} should be a parameterized string obtained from the terminfo database. E.g. \texttt{tparm(tigetstr("cup"), 5, 3)} could result in \texttt{\textbackslash 033[6;4H}, the exact result depending on terminal type.

\textbf{typeahead} \((\textit{fd})\)

Specifies that the file descriptor \textit{fd} be used for typeahead checking. If \textit{fd} is \(-1\), then no typeahead checking is done.

The curses library does “line-breakout optimization” by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or douupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

\textbf{unctrl} \((\texttt{ch})\)

Returns a string which is a printable representation of the character \texttt{ch}. Control characters are displayed as a caret followed by the character, for example as \texttt{^C}. Printing characters are left as they are.

\textbf{ungetch} \((\texttt{ch})\)

Push \texttt{ch} so the next \texttt{getch()} will return it. \textbf{Note}: only one \texttt{ch} can be pushed before \texttt{getch()} is called.

\textbf{ungetmouse} \((\texttt{id}, \texttt{x}, \texttt{y}, \texttt{z}, \texttt{bstate})\)

Push a KEY_MOUSE event onto the input queue, associating the given state data with it.

\textbf{use_env} \((\texttt{flag})\)

If used, this function should be called before \texttt{initscr()} or newterm are called. When \texttt{flag} is false, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

6.13.2 Window Objects

Window objects, as returned by \texttt{initscr()} and \texttt{newwin()} above, have the following methods:

\textbf{addch} \([(\texttt{y, x}], \texttt{ch}, \texttt{attr})\]

\textbf{Note}: A \texttt{character} means a C character (i.e., an ASCII code), rather then a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The builtin \texttt{ord()} is handy for conveying strings to codes.

Paint character \texttt{ch} at \((\texttt{y}, \texttt{x})\) with attributes \texttt{attr}, overwriting any character previously painter at that location.

By default, the character position and attributes are the current settings for the window object.

\textbf{addnstr} \([(\texttt{y, x}], \texttt{str}, \texttt{n}, \texttt{attr})\)

Paint at most \texttt{n} characters of the string \texttt{str} at \((\texttt{y}, \texttt{x})\) with attributes \texttt{attr}, overwriting anything previously on the display.

\textbf{addstr} \([(\texttt{y, x}], \texttt{str}, \texttt{attr})\)

Paint the string \texttt{str} at \((\texttt{y}, \texttt{x})\) with attributes \texttt{attr}, overwriting anything previously on the display.

\textbf{attroff} \((\texttt{attr})\)

Remove attribute \texttt{attr} from the “background” set applied to all writes to the current window.

\textbf{attron} \((\texttt{attr})\)

Add attribute \texttt{attr} from the “background” set applied to all writes to the current window.

\textbf{attrset} \((\texttt{attr})\)

Set the “background” set of attributes to \texttt{attr}. This set is initially 0 (no attributes).
**bkgd**(ch[, attr])

Sets the background property of the window to the character ch, with attributes attr. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

**bkgdset**(ch[, attr])

Sets the window’s background. A window’s background consists of a character and any combination of attributes. The attribute part of the background is combined (OR’ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

**border**([ls, rs, ts, bs, tl, tr, bl, br])

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters must be specified as integers; using one-character strings will cause `TypeError` to be raised.

**Note:** A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>Left side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>rs</td>
<td>Right side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>ts</td>
<td>Top</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>bs</td>
<td>Bottom</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>tl</td>
<td>Upper-left corner</td>
<td>ACS_ULCORNER</td>
</tr>
<tr>
<td>tr</td>
<td>Upper-right corner</td>
<td>ACS_URCORNER</td>
</tr>
<tr>
<td>bl</td>
<td>Bottom-left corner</td>
<td>ACS_BLCORNER</td>
</tr>
<tr>
<td>br</td>
<td>Bottom-right corner</td>
<td>ACS_BRCORNER</td>
</tr>
</tbody>
</table>

**box**(vertch, horch)

Similar to `border()`, but both `ls` and `rs` are `vertch` and both `ts` and `bs` are `horch`. The default corner characters are always used by this function.

**clear()**

Like `erase()`, but also causes the whole window to be repainted upon next call to `refresh()`.

**clearok**(yes)

If `yes` is 1, the next call to `refresh()` will clear the window completely.

**clrtobot()**

Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of `clrtoeol()` is performed.

**clrtoeol()**

Erase from cursor to the end of the line.

**cursyncup()**

Updates the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

**delch**([x, y])

Delete any character at (y, x).

**delete1n()**

Delete the line under the cursor. All following lines are moved up by 1 line.

**derwin**(nlines, ncols, begin_y, begin_x)

An abbreviation for “derive window”, `derwin()` is the same as calling `subwin()`, except that `begin_y` and
begin

_begin_x are relative to the origin of the window, rather than relative to the entire screen. Returns a window object for the derived window.

**echochar** (*ch[, *attr]*)
Add character *ch* with attribute *attr*, and immediately call **refresh**() on the window.

**enclose** (*y, x*)
Tests whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning true or false. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

**erase**()
Clear the window.

**getbegyx**()
Return a tuple (*y, x*) of co-ordinates of upper-left corner.

**getch** ([*x, y]*)
Get a character. Note that the integer returned does not have to be in ASCII range: function keys, keypad keys and so on return numbers higher than 256. In no-delay mode, an exception is raised if there is no input.

**getkey** ([*x, y]*)
Get a character, returning a string instead of an integer, as **getch**() does. Function keys, keypad keys and so on return a multibyte string containing the key name. In no-delay mode, an exception is raised if there is no input.

**getmaxyx**()
Return a tuple (*y, x*) of the height and width of the window.

**getparyx**()
Returns the beginning coordinates of this window relative to its parent window into two integer variables *y* and *x*. Returns −1, −1 if this window has no parent.

**getstr** ([*x, y]*)
Read a string from the user, with primitive line editing capacity.

**getyx**()
Return a tuple (*y, x*) of current cursor position relative to the window’s upper-left corner.

**hline** ([*y, x, ch, n]*)
Display a horizontal line starting at (*y, x*) with length *n* consisting of the character *ch*.

**idcok** (*flag*)
If *flag* is false, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is true, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

**idlok** (*yes*)
If called with *yes* equal to 1, curses will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

**immedok** (*flag*)
If *flag* is true, any change in the window image automatically causes the window to be refreshed; you no longer have to call **refresh**() yourself. However, it may degrade performance considerably, due to repeated calls to **wrefresh**. This option is disabled by default.

**inch** ([*x, y]*)
Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

**insch** ([*y, x, ch[, attr]*)
Paint character *ch* at (*y, x*) with attributes *attr*, moving the line from position *x* right by one character.
**insdelln(nlines)**

Inserts *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

**insertln()**

Insert a blank line under the cursor. All following lines are moved down by 1 line.

**insnstr([y, x], str, n, [attr])**

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

**instr([y, x], str, [attr])**

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

**is_linetouched(line)**

Returns true if the specified line was modified since the last call to refresh(); otherwise returns false. Raises a curses.error exception if line is not valid for the given window.

**is_wintouched()**

Returns true if the specified window was modified since the last call to refresh(); otherwise returns false.

**keypad(yes)**

If yes is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by curses. If yes is 0, escape sequences will be left as is in the input stream.

**leaveok(yes)**

If yes is 1, cursor is left where it is on update, instead of being at “cursor position.” This reduces cursor movement where possible. If possible the cursor will be made invisible.

If yes is 0, cursor will always be at “cursor position” after an update.

**move(new_y, new_x)**

Move cursor to (*new_y, new_x*).

**mvderwin(y, x)**

Moves the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

**mvwin(new_y, new_x)**

Move the window so its upper-left corner is at (*new_y, new_x*).

**nodelay(yes)**

If yes is 1, getch() will be non-blocking.

**notimeout(yes)**

If yes is 1, escape sequences will not be timed out.

If yes is 0, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

**noutrefresh()**

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call doupdate().

**6.13. curses — Terminal handling for character-cell displays**
overlay \(\text{destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol ]}\)

Overlay the window on top of \(\text{destwin}\). The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of \(\text{destwin}\).

To get fine-grained control over the copied region, the second form of \(\text{overlay()}\) can be used. \(\text{sminrow}\) and \(\text{smincol}\) are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

overwrite \(\text{destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol ]}\)

Overwrite the window on top of \(\text{destwin}\). The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of \(\text{destwin}\).

To get fine-grained control over the copied region, the second form of \(\text{overwrite()}\) can be used. \(\text{sminrow}\) and \(\text{smincol}\) are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

putwin \((\text{file})\)

Writes all data associated with the window into the provided file object. This information can be later retrieved using the \(\text{getwin()}\) function.

redrawln \((\text{beg, num})\)

Indicates that the \(\text{num}\) screen lines, starting at line \(\text{beg}\), are corrupted and should be completely redrawn on the next \(\text{refresh()}\) call.

redrawwin()

Touches the entire window, causing it to be completely redrawn on the next \(\text{refresh()}\) call.

refresh \(\left[\text{pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol}\right]\)

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with \(\text{newpad()}\). The additional parameters are needed to indicate what part of the pad and screen are involved. \(\text{pminrow}\) and \(\text{pmincol}\) specify the upper left-hand corner of the rectangle to be displayed in the pad. \(\text{sminrow}\), \(\text{smincol}\), \(\text{smaxrow}\), and \(\text{smaxcol}\) specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of \(\text{pminrow}\), \(\text{pmincol}\), \(\text{sminrow}\), \(\text{smincol}\), \(\text{smaxrow}\), or \(\text{smincol}\) are treated as if they were zero.

scroll \(\left[\text{lines = 1}\right]\)

Scroll the screen or scrolling region upward by \(\text{lines}\) lines.

scrollok \((\text{flag})\)

Controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If \(\text{flag}\) is false, the cursor is left on the bottom line. If \(\text{flag}\) is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call \(\text{idlok()}\).

setscrcrreg \((\text{top, bottom})\)

Set the scrolling region from line \(\text{top}\) to line \(\text{bottom}\). All scrolling actions will take place in this region.

standend()

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

standout()

Turn on attribute \(\text{A\_STANDOUT}\).

subpad \(\left[\text{lnlines, ncols, } (\text{begin}_y, \text{begin}_x)\right]\)

Return a sub-window, whose upper-left corner is at \((\text{begin}_y, \text{begin}_x)\), and whose width/height is \(\text{ncols/lnlines}\).
**subwin**([nlines, ncols], begin_y, begin_x)

Return a sub-window, whose upper-left corner is at \((\text{begin}_y, \text{begin}_x)\), and whose width/height is \(\text{ncols}/\text{nlines}\).

By default, the sub-window will extend from the specified position to the lower right corner of the window.

**syncdown()**

Touches each location in the window that has been touched in any of its ancestor windows. This routine is called by refresh(), so it should almost never be necessary to call it manually.

**syncok** (flag)

If called with flag set to true, then syncup() is called automatically whenever there is a change in the window.

**syncup()**

Touches all locations in ancestors of the window that have been changed in the window.

**timeout** (delay)

Sets blocking or non-blocking read behavior for the window. If delay is negative, blocking read is used, which will wait indefinitely for input. If delay is zero, then non-blocking read is used, and -1 will be returned by getch() if no input is waiting. If delay is positive, then getch() will block for delay milliseconds, and return -1 if there is still no input at the end of that time.

**touchline** (start, count)

Pretend count lines have been changed, starting with line start.

**touchwin()**

Pretend the whole window has been changed, for purposes of drawing optimizations.

**untouchwin()**

Marks all lines in the window as unchanged since the last call to refresh().

**vline** ([y, x], ch, n)

Display a vertical line starting at \((y, x)\) with length \(n\) consisting of the character \(ch\).

### 6.13.3 Constants

The curses module defines the following data members:

**ERR**

Some curses routines that return an integer, such as getch(), return ERR upon failure.

**OK**

Some curses routines that return an integer, such as napms(), return OK upon success.

**version**

A string representing the current version of the module. Also available as __version__.

Several constants are available to specify character cell attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ALTCHARSET</td>
<td>Alternate character set mode.</td>
</tr>
<tr>
<td>A_BLINK</td>
<td>Blink mode.</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Bold mode.</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Dim mode.</td>
</tr>
<tr>
<td>A_NORMAL</td>
<td>Normal attribute.</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>Standout mode.</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underline mode.</td>
</tr>
</tbody>
</table>

Keys are referred to by integer constants with names starting with ‘KEY_’. The exact keycaps available are system dependent.
<table>
<thead>
<tr>
<th>Key constant</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_MIN</td>
<td>Minimum key value</td>
</tr>
<tr>
<td>KEY_BREAK</td>
<td>Break key (unreliable)</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Down-arrow</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Up-arrow</td>
</tr>
<tr>
<td>KEY_LEFT</td>
<td>Left-arrow</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Right-arrow</td>
</tr>
<tr>
<td>KEY_HOME</td>
<td>Home key (upward+left arrow)</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Backspace (unreliable)</td>
</tr>
<tr>
<td>KEY_F0</td>
<td>Function keys. Up to 64 function keys are supported.</td>
</tr>
<tr>
<td>KEY_Fn</td>
<td>Value of function key n</td>
</tr>
<tr>
<td>KEY_DL</td>
<td>Delete line</td>
</tr>
<tr>
<td>KEY_IL</td>
<td>Insert line</td>
</tr>
<tr>
<td>KEY_DC</td>
<td>Delete character</td>
</tr>
<tr>
<td>KEY_IC</td>
<td>Insert char or enter insert mode</td>
</tr>
<tr>
<td>KEY_EIC</td>
<td>Exit insert char mode</td>
</tr>
<tr>
<td>KEY_CLEAR</td>
<td>Clear screen</td>
</tr>
<tr>
<td>KEY_EOS</td>
<td>Clear to end of screen</td>
</tr>
<tr>
<td>KEY_EOL</td>
<td>Clear to end of line</td>
</tr>
<tr>
<td>KEY_SF</td>
<td>Scroll 1 line forward</td>
</tr>
<tr>
<td>KEY_SR</td>
<td>Scroll 1 line backward (reverse)</td>
</tr>
<tr>
<td>KEY_NPAGE</td>
<td>Next page</td>
</tr>
<tr>
<td>KEY_PPAGE</td>
<td>Previous page</td>
</tr>
<tr>
<td>KEY_STAB</td>
<td>Set tab</td>
</tr>
<tr>
<td>KEY_CTAB</td>
<td>Clear tab</td>
</tr>
<tr>
<td>KEY_CATAB</td>
<td>Clear all tabs</td>
</tr>
<tr>
<td>KEY_ENTER</td>
<td>Enter or send (unreliable)</td>
</tr>
<tr>
<td>KEY_SRESET</td>
<td>Soft (partial) reset (unreliable)</td>
</tr>
<tr>
<td>KEY_RESET</td>
<td>Reset or hard reset (unreliable)</td>
</tr>
<tr>
<td>KEY_PRINT</td>
<td>Print</td>
</tr>
<tr>
<td>KEY_LL</td>
<td>Home down or bottom (lower left)</td>
</tr>
<tr>
<td>KEY_A1</td>
<td>Upper left of keypad</td>
</tr>
<tr>
<td>KEY_A3</td>
<td>Upper right of keypad</td>
</tr>
<tr>
<td>KEY_B2</td>
<td>Center of keypad</td>
</tr>
<tr>
<td>KEY_C1</td>
<td>Lower left of keypad</td>
</tr>
<tr>
<td>KEY_C3</td>
<td>Lower right of keypad</td>
</tr>
<tr>
<td>KEY_BTAB</td>
<td>Back tab</td>
</tr>
<tr>
<td>KEY_BEG</td>
<td>Beg (beginning)</td>
</tr>
<tr>
<td>KEYCANCEL</td>
<td>Cancel</td>
</tr>
<tr>
<td>KEY_CLOSE</td>
<td>Close</td>
</tr>
<tr>
<td>KEY_COMMAND</td>
<td>Cmd (command)</td>
</tr>
<tr>
<td>KEY_COPY</td>
<td>Copy</td>
</tr>
<tr>
<td>KEY_CREATE</td>
<td>Create</td>
</tr>
<tr>
<td>KEY_END</td>
<td>End</td>
</tr>
<tr>
<td>KEY_EXIT</td>
<td>Exit</td>
</tr>
<tr>
<td>KEY_FIND</td>
<td>Find</td>
</tr>
<tr>
<td>KEY_HELP</td>
<td>Help</td>
</tr>
<tr>
<td>KEY_MARK</td>
<td>Mark</td>
</tr>
<tr>
<td>KEY_MESSAGE</td>
<td>Message</td>
</tr>
<tr>
<td>KEY_MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>KEY_NEXT</td>
<td>Next</td>
</tr>
<tr>
<td>KEY_OPEN</td>
<td>Open</td>
</tr>
<tr>
<td>KEY_OPTIONS</td>
<td>Options</td>
</tr>
<tr>
<td>KEY_PREVIOUS</td>
<td>Prev (previous)</td>
</tr>
<tr>
<td>Key constant</td>
<td>Key</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>KEY_REDO</td>
<td>Redo</td>
</tr>
<tr>
<td>KEY_REFERENCE</td>
<td>Ref (reference)</td>
</tr>
<tr>
<td>KEY_REFRESH</td>
<td>Refresh</td>
</tr>
<tr>
<td>KEY_REPLACE</td>
<td>Replace</td>
</tr>
<tr>
<td>KEY_RESTART</td>
<td>Restart</td>
</tr>
<tr>
<td>KEY_RESUME</td>
<td>Resume</td>
</tr>
<tr>
<td>KEY_SAVE</td>
<td>Save</td>
</tr>
<tr>
<td>KEY_SBE</td>
<td>Shifted Beg (beginning)</td>
</tr>
<tr>
<td>KEY_SCANCEL</td>
<td>Shifted Cancel</td>
</tr>
<tr>
<td>KEY_SCOMMAND</td>
<td>Shifted Command</td>
</tr>
<tr>
<td>KEY_SCOPY</td>
<td>Shifted Copy</td>
</tr>
<tr>
<td>KEY_SCREATE</td>
<td>Shifted Create</td>
</tr>
<tr>
<td>KEY_SDC</td>
<td>Shifted Delete char</td>
</tr>
<tr>
<td>KEY(SDL</td>
<td>Shifted Delete line</td>
</tr>
<tr>
<td>KEY_SELECT</td>
<td>Select</td>
</tr>
<tr>
<td>KEY_SEND</td>
<td>Shifted End</td>
</tr>
<tr>
<td>KEY_SEOL</td>
<td>Shifted Clear line</td>
</tr>
<tr>
<td>KEY_SEXT</td>
<td>Shifted Dxit</td>
</tr>
<tr>
<td>KEY_SFIND</td>
<td>Shifted Find</td>
</tr>
<tr>
<td>KEY_SHELP</td>
<td>Shifted Help</td>
</tr>
<tr>
<td>KEY_SHOME</td>
<td>Shifted Home</td>
</tr>
<tr>
<td>KEY_SIC</td>
<td>Shifted Input</td>
</tr>
<tr>
<td>KEY_SLEFT</td>
<td>Shifted Left arrow</td>
</tr>
<tr>
<td>KEY_SMESSAGE</td>
<td>Shifted Message</td>
</tr>
<tr>
<td>KEY_SMOVE</td>
<td>Shifted Move</td>
</tr>
<tr>
<td>KEY_SNEXT</td>
<td>Shifted Next</td>
</tr>
<tr>
<td>KEY_SOPTIONS</td>
<td>Shifted Options</td>
</tr>
<tr>
<td>KEY_SPREVIOUS</td>
<td>Shifted Prev</td>
</tr>
<tr>
<td>KEY_SPRINT</td>
<td>Shifted Print</td>
</tr>
<tr>
<td>KEY_SREDO</td>
<td>Shifted Redo</td>
</tr>
<tr>
<td>KEY_SREPLACE</td>
<td>Shifted Replace</td>
</tr>
<tr>
<td>KEY_SRIGHT</td>
<td>Shifted Right arrow</td>
</tr>
<tr>
<td>KEY_SRSUME</td>
<td>Shifted Resume</td>
</tr>
<tr>
<td>KEY_SSAVE</td>
<td>Shifted Save</td>
</tr>
<tr>
<td>KEY_SSUSPEND</td>
<td>Shifted Suspend</td>
</tr>
<tr>
<td>KEY_UNDO</td>
<td>Shifted Undo</td>
</tr>
<tr>
<td>KEY_SUSPEND</td>
<td>Suspend</td>
</tr>
<tr>
<td>KEY_MOUSE</td>
<td>Mouse event has occurred</td>
</tr>
<tr>
<td>KEY_RESIZE</td>
<td>Terminal resize event</td>
</tr>
<tr>
<td>KEY_MAX</td>
<td>Maximum key value</td>
</tr>
</tbody>
</table>

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:
The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation. **Note:** These are available only after `initscr()` has been called.

<table>
<thead>
<tr>
<th>ACS code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS_BBSS</td>
<td>alternate name for upper right corner</td>
</tr>
<tr>
<td>ACS_BLOCK</td>
<td>solid square block</td>
</tr>
<tr>
<td>ACS_BOARD</td>
<td>board of squares</td>
</tr>
<tr>
<td>ACS_BSBS</td>
<td>alternate name for horizontal line</td>
</tr>
<tr>
<td>ACS_BSSB</td>
<td>alternate name for upper left corner</td>
</tr>
<tr>
<td>ACS_BSSS</td>
<td>alternate name for top tee</td>
</tr>
<tr>
<td>ACS_BTEE</td>
<td>bottom tee</td>
</tr>
<tr>
<td>ACS_BULLET</td>
<td>bullet</td>
</tr>
<tr>
<td>ACS_CKBOARD</td>
<td>checker board (stipple)</td>
</tr>
<tr>
<td>ACS_DARROW</td>
<td>arrow pointing down</td>
</tr>
<tr>
<td>ACS_DEGREE</td>
<td>degree symbol</td>
</tr>
<tr>
<td>ACS_DIAMOND</td>
<td>diamond</td>
</tr>
<tr>
<td>ACS_GEQUAL</td>
<td>greater-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_HLINE</td>
<td>horizontal line</td>
</tr>
<tr>
<td>ACS_LANTERN</td>
<td>lantern symbol</td>
</tr>
<tr>
<td>ACS_LARROW</td>
<td>left arrow</td>
</tr>
<tr>
<td>ACS_LEQUAL</td>
<td>less-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_LLCORNER</td>
<td>lower left-hand corner</td>
</tr>
<tr>
<td>ACS_LRCORNER</td>
<td>lower right-hand corner</td>
</tr>
<tr>
<td>ACS_LTEE</td>
<td>left tee</td>
</tr>
<tr>
<td>ACS_NEQUAL</td>
<td>not-equal sign</td>
</tr>
<tr>
<td>ACS_PI</td>
<td>letter pi</td>
</tr>
<tr>
<td>ACS_PMINUS</td>
<td>plus-or-minus sign</td>
</tr>
<tr>
<td>ACS_PLUS</td>
<td>big plus sign</td>
</tr>
<tr>
<td>ACS_RARROW</td>
<td>right arrow</td>
</tr>
<tr>
<td>ACS_RTEE</td>
<td>right tee</td>
</tr>
<tr>
<td>ACS_S1</td>
<td>scan line 1</td>
</tr>
<tr>
<td>ACS_S3</td>
<td>scan line 3</td>
</tr>
<tr>
<td>ACS_S7</td>
<td>scan line 7</td>
</tr>
<tr>
<td>ACS_S9</td>
<td>scan line 9</td>
</tr>
<tr>
<td>ACS_SBBS</td>
<td>alternate name for lower right corner</td>
</tr>
<tr>
<td>ACS_SBSB</td>
<td>alternate name for vertical line</td>
</tr>
<tr>
<td>ACS_SSSB</td>
<td>alternate name for right tee</td>
</tr>
<tr>
<td>ACS_SSSB</td>
<td>alternate name for lower left corner</td>
</tr>
<tr>
<td>ACS_SSSS</td>
<td>alternate name for bottom tee</td>
</tr>
<tr>
<td>ACS_SSSS</td>
<td>alternate name for left tee</td>
</tr>
<tr>
<td>ACS_STERLING</td>
<td>pound sterling</td>
</tr>
<tr>
<td>ACS_TTEE</td>
<td>top tee</td>
</tr>
<tr>
<td>ACS_UARROW</td>
<td>up arrow</td>
</tr>
<tr>
<td>ACS_ULCORNER</td>
<td>upper left corner</td>
</tr>
</tbody>
</table>
The following table lists the predefined colors:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_BLACK</td>
<td>Black</td>
</tr>
<tr>
<td>COLOR_BLUE</td>
<td>Blue</td>
</tr>
<tr>
<td>COLOR_CYAN</td>
<td>Cyan (light greenish blue)</td>
</tr>
<tr>
<td>COLOR_GREEN</td>
<td>Green</td>
</tr>
<tr>
<td>COLOR_MAGENTA</td>
<td>Magenta (purplish red)</td>
</tr>
<tr>
<td>COLOR_RED</td>
<td>Red</td>
</tr>
<tr>
<td>COLOR_WHITE</td>
<td>White</td>
</tr>
<tr>
<td>COLOR_YELLOW</td>
<td>Yellow</td>
</tr>
</tbody>
</table>

6.14 `curses.textpad` — Text input widget for curses programs

New in version 1.6.

The `curses.textpad` module provides a `Textbox` class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function:

```python
rectangle(win, uly, ulx, lry, lrx)
```

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

6.14.1 Textbox objects

You can instantiate a `Textbox` object as follows:

```python
class Textbox(win)
```

Return a textbox widget object. The `win` argument should be a curses `WindowObject` in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's `stripspaces` flag is initially on.

Textbox objects have the following methods:

```python
edit([validator])
```

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If `validator` is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` member.
**do_command**(ch)

Process a single command keystroke. Here are the supported special keystrokes:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-A</td>
<td>Go to left edge of window.</td>
</tr>
<tr>
<td>Ctrl-B</td>
<td>Cursor left, wrapping to previous line if appropriate.</td>
</tr>
<tr>
<td>Ctrl-D</td>
<td>Delete character under cursor.</td>
</tr>
<tr>
<td>Ctrl-E</td>
<td>Go to right edge (stripspaces off) or end of line (stripspaces on).</td>
</tr>
<tr>
<td>Ctrl-F</td>
<td>Cursor right, wrapping to next line when appropriate.</td>
</tr>
<tr>
<td>Ctrl-G</td>
<td>Terminate, returning the window contents.</td>
</tr>
<tr>
<td>Ctrl-H</td>
<td>Delete character backward.</td>
</tr>
<tr>
<td>Ctrl-J</td>
<td>Terminate if the window is 1 line, otherwise insert newline.</td>
</tr>
<tr>
<td>Ctrl-K</td>
<td>If line is blank, delete it, otherwise clear to end of line.</td>
</tr>
<tr>
<td>Ctrl-L</td>
<td>Refresh screen.</td>
</tr>
<tr>
<td>Ctrl-N</td>
<td>Cursor down; move down one line.</td>
</tr>
<tr>
<td>Ctrl-O</td>
<td>Insert a blank line at cursor location.</td>
</tr>
<tr>
<td>Ctrl-P</td>
<td>Cursor up; move up one line.</td>
</tr>
</tbody>
</table>

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_LEFT</td>
<td>Ctrl-B</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Ctrl-F</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Ctrl-P</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Ctrl-N</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Ctrl-h</td>
</tr>
</tbody>
</table>

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

**gather()**

This method returns the window contents as a string; whether blanks in the window are included is affected by the **stripspaces** member.

**stripspaces**

This data member is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents is gathered.

## 6.15 curses.wrapper — Terminal handler for curses programs

New in version 1.6.

This module supplies one function, **wrapper()**, which runs another function which should be the rest of your curses-using application. If the application raises an exception, **wrapper()** will restore the terminal to a sane state before passing it further up the stack and generating a traceback.

**wrapper**(func, ...)

Wrapper function that initializes curses and calls another function, **func**, restoring normal keyboard/screen behavior on error. The callable object **func** is then passed the main window `stdscr` as its first argument, followed by any other arguments passed to **wrapper()**.

Before calling the hook function, **wrapper()** turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.
6.16 curses.ascii — Utilities for ASCII characters

New in version 1.6.

The curses.ascii module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>Start of heading, console interrupt</td>
</tr>
<tr>
<td>SOH</td>
<td>Start of text</td>
</tr>
<tr>
<td>STX</td>
<td>End of text</td>
</tr>
<tr>
<td>ETX</td>
<td>End of transmission</td>
</tr>
<tr>
<td>EOT</td>
<td>End of transmission</td>
</tr>
<tr>
<td>ENQ</td>
<td>Enquiry, goes with ACK flow control</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>BEL</td>
<td>Bell</td>
</tr>
<tr>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>TAB</td>
<td>Tab</td>
</tr>
<tr>
<td>HT</td>
<td>Alias for TAB: “Horizontal tab”</td>
</tr>
<tr>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>NL</td>
<td>Alias for LF: “New line”</td>
</tr>
<tr>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>SO</td>
<td>Shift-out, begin alternate character set</td>
</tr>
<tr>
<td>SI</td>
<td>Shift-in, resume default character set</td>
</tr>
<tr>
<td>DLE</td>
<td>Data-link escape</td>
</tr>
<tr>
<td>DC1</td>
<td>XON, for flow control</td>
</tr>
<tr>
<td>DC2</td>
<td>Device control 2, block-mode flow control</td>
</tr>
<tr>
<td>DC4</td>
<td>Device control 4</td>
</tr>
<tr>
<td>NAK</td>
<td>Negative acknowledgement</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>ETB</td>
<td>End transmission block</td>
</tr>
<tr>
<td>CAN</td>
<td>Cancel</td>
</tr>
<tr>
<td>EM</td>
<td>End of medium</td>
</tr>
<tr>
<td>SUB</td>
<td>Substitute</td>
</tr>
<tr>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>FS</td>
<td>File separator</td>
</tr>
<tr>
<td>GS</td>
<td>Group separator</td>
</tr>
<tr>
<td>RS</td>
<td>Record separator, block-mode terminator</td>
</tr>
<tr>
<td>US</td>
<td>Unit separator</td>
</tr>
<tr>
<td>SP</td>
<td>Space</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

**isalnum(c)**
Checks for an ASCII alphanumeric character; it is equivalent to ‘isalpha(c) or isdigit(c)’.

**isalpha(c)**
Checks for an ASCII alphabetic character; it is equivalent to ‘isupper(c) or islower(c)’.

isascii(c)
Checks for a character value that fits in the 7-bit ASCII set.

isblank(c)
Checks for an ASCII whitespace character.

iscntrl(c)
Checks for an ASCII control character (in the range 0x00 to 0x1f).

isdigit(c)
Checks for an ASCII decimal digit, ‘0’ through ‘9’. This is equivalent to ‘c in string.digits’.

isgraph(c)
Checks for ASCII any printable character except space.

islower(c)
Checks for an ASCII lower-case character.

isprint(c)
Checks for any ASCII printable character including space.

ispunct(c)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

isspace(c)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

isupper(c)
Checks for an ASCII uppercase letter.

isxdigit(c)
Checks for an ASCII hexadecimal digit. This is equivalent to ‘c in string.hexdigits’.

ismeta(c)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or strings; when the argument is a string, it is first converted using the built-in function ord().

Note that all these functions check ordinal bit values derived from the first character of the string you pass in; they do not actually know anything about the host machine’s character encoding. For functions that know about the character encoding (and handle internationalization properly) see the string module.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

ascii(c)
Return the ASCII value corresponding to the low 7 bits of c.

ctrl(c)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

alt(c)
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

unctrl(c)
Return a string representation of the ASCII character \( c \). If \( c \) is printable, this string is the character itself. If the character is a control character (0x00-0x1f) the string consists of a caret (‘^’) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is ‘ˆ?’ . If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and ‘!’ prepended to the result.

controlnames
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic ‘SP’ for the space character.

6.17 curses.panel — A panel stack extension for curses.

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

6.17.1 Functions

The module curses.panel defines the following functions:

bottom_panel ()

Returns the bottom panel in the panel stack.

new_panel (win)

Returns a panel object, associating it with the given window \( \text{win} \).

top_panel ()

Returns the top panel in the panel stack.

update_panels ()

Updates the virtual screen after changes in the panel stack. This does not call curses.doupdate (), so you’ll have to do this yourself.

6.17.2 Panel Objects

Panel objects, as returned by new_panel () above, are windows with a stacking order. There’s always a window associated with a panel which determines the content, while the panel methods are responsible for the window’s depth in the panel stack.

Panel objects have the following methods:

above ()

Returns the panel above the current panel.

dbelow ()

Returns the panel below the current panel.

dbottom ()

Push the panel to the bottom of the stack.

hidden ()

Returns true if the panel is hidden (not visible), false otherwise.

hide ()

Hide the panel. This does not delete the object, it just makes the window on screen invisible.

move (y, x)

Move the panel to the screen coordinates \( (y, x) \).
replace\( (\text{win}) \)
Change the window associated with the panel to the window \( \text{win} \).

\textbf{set_userptr\( (\text{obj}) \)}
Set the panel’s user pointer to \( \text{obj} \). This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

\textbf{show\( () \)}
Display the panel (which might have been hidden).

\textbf{top\( () \)}
Push panel to the top of the stack.

\textbf{userptr\( () \)}
Returns the user pointer for the panel. This might be any Python object.

\textbf{window\( () \)}
Returns the window object associated with the panel.

\section{6.18 \texttt{getopt} — Parser for command line options}

This module helps scripts to parse the command line arguments in \texttt{sys.argv}. It supports the same conventions as the UNIX \texttt{getopt()} function (including the special meanings of arguments of the form ‘-’ and ‘--’). Long options similar to those supported by GNU software may be used as well via an optional third argument. This module provides a single function and an exception:

\textbf{getopt\( ([\text{args}, \text{options}], \text{long_options}]) \)}
Parses command line options and parameter list. \texttt{args} is the argument list to be parsed, without the leading reference to the running program. Typically, this means \texttt{sys.argv[1:]} . \texttt{options} is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (‘:’; i.e., the same format that UNIX \texttt{getopt()} uses).

\textbf{Note:} Unlike GNU \texttt{getopt()}, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU UNIX systems work.

\texttt{long_options}, if specified, must be a list of strings with the names of the long options which should be supported. The leading ‘--’ characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (‘=’). To accept only long options, \texttt{options} should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if \texttt{long_options} is [‘\text{foo}', 'frob'], the option \texttt{--fo} will match as \texttt{--foo}, but \texttt{--f} will not match uniquely, so \texttt{GetoptError} will be raised.

The return value consists of two elements: the first is a list of \texttt{(option, value)} pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of \texttt{args}). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., ‘-x’) or two hyphens for long options (e.g., ‘--long-option’), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

\textbf{exception \texttt{GetoptError}}
This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes \texttt{msg} and \texttt{opt} give the error message and related option; if there is no specific option to which the exception relates, \texttt{opt} is an empty string.

\textbf{exception \texttt{error}}
Alias for \texttt{GetoptError}; for backward compatibility.
An example using only UNIX style options:

```python
>>> import getopt
>>> args = '-a -b -cfoo -d bar al a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'al', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d: ')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```python
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', ['condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

In a script, typical usage is something like this:

```python
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:", ["help", "output="])  
    except getopt.GetoptError:
        # print help information and exit
        usage()
        sys.exit(2)
        output = None
    for o, a in opts:
        if o in ('--help', '-h'):
            usage()
            sys.exit()
        if o in ('--output', '-o'):
            output = a
            # ...

if __name__ == '__main__':
    main()
```

6.18. getopt — Parser for command line options
6.19  tempfile — Generate temporary file names

This module generates temporary file names. It is not UNIX specific, but it may require some help on non-UNIX systems.

The module defines the following user-callable functions:

\[ \text{mktemp}([\text{suffix}]) \]

Return a unique temporary filename. This is an absolute pathname of a file that does not exist at the time the call is made. No two calls will return the same filename. suffix, if provided, is used as the last part of the generated file name. This can be used to provide a filename extension or other identifying information that may be useful on some platforms.

\[ \text{TemporaryFile}([\text{mode}[\text{bufsize}, \text{suffix}]]) \]

Return a file (or file-like) object that can be used as a temporary storage area. The file is created in the most secure manner available in the appropriate temporary directory for the host platform. Under UNIX, the directory entry to the file is removed so that it is secure against attacks which involve creating symbolic links to the file or replacing the file with a symbolic link to some other file. For other platforms, which don’t allow removing the directory entry while the file is in use, the file is automatically deleted as soon as it is closed (including an implicit close when it is garbage-collected).

The mode parameter defaults to ‘w+b’ so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. bufsize defaults to –1, meaning that the operating system default is used. suffix is passed to mktemp().

The module uses two global variables that tell it how to construct a temporary name. The caller may assign values to them; by default they are initialized at the first call to mktemp().

\[ \text{tempdir} \]

When set to a value other than None, this variable defines the directory in which filenames returned by mktemp() reside. The default is taken from the environment variable TMPDIR; if this is not set, either ‘/usr/tmp’ is used (on UNIX), or the current working directory (all other systems). No check is made to see whether its value is valid.

\[ \text{gettempprefix}() \]

Return the filename prefix used to create temporary files. This does not contain the directory component. Using this function is preferred over using the template variable directly. New in version 1.5.2.

\[ \text{template} \]

Deprecated since release 2.0. Use gettempprefix() instead.

When set to a value other than None, this variable defines the prefix of the final component of the filenames returned by mktemp(). A string of decimal digits is added to generate unique filenames. The default is either ‘@pid.’ where pid is the current process ID (on UNIX), ‘~pid-’ on Windows NT, ‘Python-Tmp-’ on MacOS, or ‘tmp’ (all other systems).

Older versions of this module used to require that template be set to None after a call to os.fork(); this has not been necessary since version 1.5.2.

6.20  errno — Standard errno system symbols

This module makes available standard errno system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from ‘linux/include/errno.h’, which should be pretty all-inclusive.

\[ \text{errorcode} \]

Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, errno.errorcode[errno.EPERM] maps to ‘EPERM’.

To translate a numeric error code to an error message, use os.strerror().
Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

- **EPERM**  
  Operation not permitted

- **ENOENT**  
  No such file or directory

- **ESRCH**  
  No such process

- **EINTR**  
  Interrupted system call

- **EIO**  
  I/O error

- **ENXIO**  
  No such device or address

- **E2BIG**  
  Arg list too long

- **ENOEXEC**  
  Exec format error

- **EBADF**  
  Bad file number

- **ECHILD**  
  No child processes

- **EAGAIN**  
  Try again

- **ENOMEM**  
  Out of memory

- **EACCES**  
  Permission denied

- **EFAULT**  
  Bad address

- **ENOTBLK**  
  Block device required

- **EBUSY**  
  Device or resource busy

- **EEXIST**  
  File exists

- **EXDEV**  
  Cross-device link

- **ENODEV**  
  No such device

- **ENOTDIR**  
  Not a directory

- **EISDIR**  
  Is a directory
EINVAL
   Invalid argument
ENFILE
   File table overflow
EMFILE
   Too many open files
ENOTTY
   Not a typewriter
ETXTBSY
   Text file busy
EFBIG
   File too large
ENOSPC
   No space left on device
ESPIPE
   Illegal seek
EROFS
   Read-only file system
EMLINK
   Too many links
EPIPE
   Broken pipe
EDOM
   Math argument out of domain of func
ERANGE
   Math result not representable
EDEADLK
   Resource deadlock would occur
ENAMETOOLONG
   File name too long
ENOLCK
   No record locks available
ENOSYS
   Function not implemented
ENOTEMPTY
   Directory not empty
ELOOP
   Too many symbolic links encountered
EWADDLACK
   Operation would block
ENOMSG
   No message of desired type
EIDRM
   Identifier removed
**ECHRNG**
Channel number out of range

**EL2NSYNC**
Level 2 not synchronized

**EL3HLT**
Level 3 halted

**EL3RST**
Level 3 reset

**ELNRNG**
Link number out of range

**EUNATCH**
Protocol driver not attached

**ENOCSI**
No CSI structure available

**EL2HLT**
Level 2 halted

**EBADE**
Invalid exchange

**EBADR**
Invalid request descriptor

**EXFULL**
Exchange full

**ENOANO**
No anode

**EBADRQC**
Invalid request code

**EBADSLT**
Invalid slot

**EDEADLOCK**
File locking deadlock error

**EBFONT**
Bad font file format

**ENOSTR**
Device not a stream

**ENODATA**
No data available

**ETIME**
Timer expired

**ENOSR**
Out of streams resources

**ENONET**
Machine is not on the network

**ENOPKG**
Package not installed
EREMOTE
Object is remote

ENOLINK
Link has been severed

EADV
Advertise error

ESRMNT
Srmount error

ECOMM
Communication error on send

EPROTO
Protocol error

EMULTIHOP
Multihop attempted

EDOTDOT
RFS specific error

EBADMSG
Not a data message

EOVERFLOW
Value too large for defined data type

ENOTUNIQ
Name not unique on network

EBADFD
File descriptor in bad state

EREMCHG
Remote address changed

ELIBACC
Can not access a needed shared library

ELIBBAD
Accessing a corrupted shared library

ELIBSCN
.lib section in a.out corrupted

ELIBMAX
Attempting to link in too many shared libraries

ELIBEXEC
Cannot exec a shared library directly

EILSEQ
Illegal byte sequence

ERESTART
Interrupted system call should be restarted

ESTRPIPE
Streams pipe error

EUSERS
Too many users
ENOTSOCK
Socket operation on non-socket

EDESTADDRREQ
Destination address required

EMSGSIZE
Message too long

EPROTOTYPE
Protocol wrong type for socket

ENOPROTOOPT
Protocol not available

EPROTONOSUPPORT
Protocol not supported

ESOCKTNOSUPPORT
Socket type not supported

EOPNOTSUPP
Operation not supported on transport endpoint

EPFNOSUPPORT
Protocol family not supported

EAFNOSUPPORT
Address family not supported by protocol

EADDRINUSE
Address already in use

EADDRNOTAVAIL
Cannot assign requested address

ENETDOWN
Network is down

ENETUNREACH
Network is unreachable

ENETRESET
Network dropped connection because of reset

ECONNABORTED
Software caused connection abort

ECONNRESET
Connection reset by peer

ENOBUFS
No buffer space available

EISCONN
Transport endpoint is already connected

ENOTCONN
Transport endpoint is not connected

ESHUTDOWN
Cannot send after transport endpoint shutdown

ETOOMANYREFS
Too many references: cannot splice
The \texttt{glob} module finds all the pathnames matching a specified pattern according to the rules used by the UNIX shell. No tilde expansion is done, but \texttt{*, ?, and character ranges expressed with [ ]} will be correctly matched. This is done by using the \texttt{os.listdir()} and \texttt{fnmatch.fnmatch()} functions in concert, and not by actually invoking a subshell. (For tilde and shell variable expansion, use \texttt{os.path.expanduser()} and \texttt{os.path.expandvars()}.)

\begin{verbatim}
\texttt{glob}( pathname ) \\
Returns a possibly-empty list of path names that match \texttt{pathname}, which must be a string containing a path specification. \texttt{pathname} can be either absolute (like \\
'\texttt{/usr/src/Python-1.5/Makefile}') or relative (like \\
'\texttt{../Tools/*/*.gif}'), and can contain shell-style wildcards.
\end{verbatim}

For example, consider a directory containing only the following files: '1.gif', '2.txt', and 'card.gif'. \texttt{glob()} will produce the following results. Notice how any leading components of the path are preserved.
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']

See Also:
Module fnmatch (section 6.22):
    Shell-style filename (not path) expansion

6.22 fnmatch — UNIX filename pattern matching

This module provides support for UNIX shell-style wildcards, which are not the same as regular expressions (which are documented in the re module). The special characters used in shell-style wildcards are:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches everything</td>
</tr>
<tr>
<td>?</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[seq]</td>
<td>matches any character in seq</td>
</tr>
<tr>
<td>[!seq]</td>
<td>matches any character not in seq</td>
</tr>
</tbody>
</table>

Note that the filename separator (‘/’ on UNIX) is not special to this module. See module glob for pathname expansion (glob uses fnmatch() to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the * and ? patterns.

fnmatch (filename, pattern)
Test whether the filename string matches the pattern string, returning true or false. If the operating system is case-insensitive, then both parameters will be normalized to all lower- or upper-case before the comparison is performed. If you require a case-sensitive comparison regardless of whether that’s standard for your operating system, use fnmatchcase() instead.

fnmatchcase (filename, pattern)
Test whether filename matches pattern, returning true or false; the comparison is case-sensitive.

See Also:
Module glob (section 6.21):
    UNIX shell-style path expansion.

6.23 shutil — High-level file operations

The shutil module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal.

Caveat: On MacOS, the resource fork and other metadata are not used. For file copies, this means that resources will be lost and file type and creator codes will not be correct.

copyfile (src, dst)
Copy the contents of the file named src to a file named dst. If dst exists, it will be replaced, otherwise it will be created.
copyfileobj (fsrc, fdst[, length])
Copy the contents of the file-like object fsrc to the file-like object fdst. The integer length, if given, is the buffer size. In particular, a negative length value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption.

copymode (src, dst)
Copy the permission bits from src to dst. The file contents, owner, and group are unaffected.

copystat (src, dst)
Copy the permission bits, last access time, and last modification time from src to dst. The file contents, owner, and group are unaffected.

copy (src, dst)
Copy the file src to the file or directory dst. If dst is a directory, a file with the same basename as src is created (or overwritten) in the directory specified. Permission bits are copied.

copy2 (src, dst)
Similar to copy (), but last access time and last modification time are copied as well. This is similar to the UNIX command cp -p.

copytree (src, dst[, symlinks ])
Recursively copy an entire directory tree rooted at src. The destination directory, named by dst, must not already exist; it will be created. Individual files are copied using copy2 (). If symlinks is true, symbolic links in the source tree are represented as symbolic links in the new tree; if false or omitted, the contents of the linked files are copied to the new tree. Errors are reported to standard output.

The source code for this should be considered an example rather than a tool.

rmtree (path[, ignore_errors[, onerror ]])
Delete an entire directory tree. If ignore_errors is true, errors will be ignored; if false or omitted, errors are handled by calling a handler specified by onerror or raise an exception.

If onerror is provided, it must be a callable that accepts three parameters: function, path, and excinfo. The first parameter, function, is the function which raised the exception; it will be os.remove () or os.rmdir (). The second parameter, path, will be the path name passed to function. The third parameter, excinfo, will be the exception information return by sys.exc_info (). Exceptions raised by onerror will not be caught.

6.23.1 Example

This example is the implementation of the copytree () function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.
def copytree(src, dst, symlinks=0):
    names = os.listdir(src)
    os.mkdir(dst)
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname)
            else:
                copy2(srcname, dstname)
        except (IOError, os.error), why:
            print "Can't copy %s to %s: %s" % ('srcname', 'dstname', str(why))

6.24 locale — Internationalization services

The locale module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The locale module is implemented on top of the _locale module, which in turn uses an ANSI C locale implementation if available.

The locale module defines the following exception and functions:

**exception Error**

Exception raised when setlocale() fails.

**setlocale**(category[, locale])

If locale is specified, it may be a string, a tuple of the form (language code, encoding), or None. If it is a tuple, it is converted to a string using the locale aliasing engine. If locale is given and not None, setlocale() modifies the locale setting for the category. The available categories are listed in the data description below. The value is the name of a locale. An empty string specifies the user’s default settings. If the modification of the locale fails, the exception Error is raised. If successful, the new locale setting is returned.

If locale is omitted or None, the current setting for category is returned.

setlocale() is not thread safe on most systems. Applications typically start with a call of

import locale
locale.setlocale(locale.LC_ALL, '')

This sets the locale for all categories to the user’s default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

Changed in version 2.0: Added support for tuple values of the locale parameter.

**localeconv()**

Returns the database of of the local conventions as a dictionary. This dictionary has the following strings as keys:
The possible values for ‘p_sign_posn’ and ‘n_sign_posn’ are given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Currency and value are surrounded by parentheses.</td>
</tr>
<tr>
<td>1</td>
<td>The sign should precede the value and currency symbol.</td>
</tr>
<tr>
<td>2</td>
<td>The sign should follow the value and currency symbol.</td>
</tr>
<tr>
<td>3</td>
<td>The sign should immediately precede the value.</td>
</tr>
<tr>
<td>4</td>
<td>The sign should immediately follow the value.</td>
</tr>
<tr>
<td>LC_MAX</td>
<td>Nothing is specified in this locale.</td>
</tr>
</tbody>
</table>

getdefaultlocale([envvars])

Tries to determine the default locale settings and returns them as a tuple of the form (language code, encoding).

According to POSIX, a program which has not called setlocale(LC_ALL, '') runs using the portable ‘C’ locale. Calling setlocale(LC_ALL, '') lets it use the default locale as defined by the LANG variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the LANG variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. envvars defaults to the search path used in GNU gettext; it must always contain the variable name ‘LANG’. The GNU gettext search path contains ‘LANGUAGE’, ‘LC_ALL’, code’LC_CTYPE’, and ‘LANG’, in that order.

Except for the code ‘C’, the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined. New in version 2.0.

getlocale([category])

Returns the current setting for the given locale category as tuple (language code, encoding). category may be one of the LC_* values except LC_ALL. It defaults to LC_CTYPE.

Except for the code ‘C’, the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined. New in version 2.0.

normalize(localename)

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with setlocale(). If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like setlocale(). New in version 2.0.
```
resetlocale([category])

Sets the locale for category to the default setting.

The default setting is determined by calling getdefaultlocale(). category defaults to LC_ALL. New in version 2.0.

strcoll(string1, string2)

Compares two strings according to the current LC_COLLATE setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether string1 collates before or after string2 or is equal to it.

strxfrm(string)

Transforms a string to one that can be used for the built-in function cmp(), and still returns locale-aware results. This function can be used when the same string is compared repeatedly, e.g., when collating a sequence of strings.

format(format, val[, grouping])

Formats a number val according to the current LC_NUMERIC setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If grouping is true, also takes the grouping into account.

str(float)

Formats a floating point number using the same format as the built-in function str(float), but takes the decimal point into account.

atof(string)

Converts a string to a floating point number, following the LC_NUMERIC settings.

atoi(string)

Converts a string to an integer, following the LC_NUMERIC conventions.

LC_CTYPE

Locale category for the character type functions. Depending on the settings of this category, the functions of module string dealing with case change their behaviour.

LC_COLLATE

Locale category for sorting strings. The functions strcoll() and strxfrm() of the locale module are affected.

LC_TIME

Locale category for the formatting of time. The function time.strftime() follows these conventions.

LC_MONETARY

Locale category for formatting of monetary values. The available options are available from the localeconv() function.

LC_MESSAGES

Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by os.strerror() might be affected by this category.

LC_NUMERIC

Locale category for formatting numbers. The functions format(), atoi(), atof() and str() of the locale module are affected by that category. All other numeric formatting operations are not affected.

LC_ALL

Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

CHAR_MAX

This is a symbolic constant used for different values returned by localeconv().
```
6.24.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the ‘C’ locale, no matter what the user’s preferred locale is. The program must explicitly say that it wants the user’s preferred locale settings by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call `setlocale()` in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (e.g. `string.lower()`, or certain formats used with `time.strftime()`), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-‘C’ locale settings.

The case conversion functions in the `string` and `strop` modules are affected by the locale settings. When a call to the `setlocale()` function changes the LC_CTYPE settings, the variables `string.lowercase`, `string.uppercase`, and `string.letters` (and their counterparts in `strop`) are recalculated. Note that this code that uses these variable through ‘from ... import ...’, e.g. from `string import letters`, is not affected by subsequent `setlocale()` calls.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: `atof()`, `atoi()`, `format()`, `str()`.

6.24.2 For extension writers and programs that embed Python

Extension modules should never call `setlocale()`, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is ‘C’).

When Python is embedded in an application, if the application sets the locale to something specific before initializing Python, that is generally okay, and Python will use whatever locale is set, except that the LC_NUMERIC locale should always be ‘C’.

The `setlocale()` function in the `locale` module gives the Python programmer the impression that you can manipulate the LC_NUMERIC locale setting, but this not the case at the C level: C code will always find that the LC_NUMERIC locale setting is ‘C’. This is because too much would break when the decimal point character is set to something else than a period (e.g. the Python parser would break). Caveat: threads that run without holding Python’s global interpreter lock may occasionally find that the numeric locale setting differs; this is because the only portable way to implement this feature is to set the numeric locale settings to what the user requests, extract the relevant characteristics, and then restore the ‘C’ numeric locale.
When Python code uses the `locale` module to change the locale, this also affects the embedding application. If the embedding application doesn’t want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the ‘config.c’ file, and make sure that the `_locale` module is not accessible as a shared library.

6.25 gettext — Multilingual internationalization services

The `gettext` module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

6.25.1 GNU gettext API

The `gettext` module defines the following API, which is very similar to the GNU `gettext` API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

```python
bindtextdomain (domain[, localedir])
    Bind the domain to the locale directory localedir. More concretely, gettext will look for binary `.mo` files for the given domain using the path (on UNIX): `localedir/languagELC_MESSAGES/domain.mo`, where languages is searched for in the environment variables LANGUAGE, LC_ALL, LC_MESSAGES, and LANG respectively.
    If localedir is omitted or None, then the current binding for domain is returned.

textdomain ([domain])
    Change or query the current global domain. If domain is None, then the current global domain is returned, otherwise the global domain is set to domain, which is returned.

gettext (message)
    Return the localized translation of message, based on the current global domain, language, and locale directory.
    This function is usually aliased as _ in the local namespace (see examples below).

dgettext (domain, message)
    Like gettext(), but look the message up in the specified domain.
```

Note that GNU `gettext` also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here’s an example of typical usage for this API:

```python
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
ggettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

---

2The default locale directory is system dependent; e.g. on Red Hat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The `gettext` module does not try to support these system dependent defaults; instead its default is `sys.prefix/share/locale`. For this reason, it is always best to call `bindtextdomain()` with an explicit absolute path at the start of your application.
6.25.2 Class-based API

The class-based API of the gettext module gives you more flexibility and greater convenience than the GNU gettext API. It is the recommended way of localizing your Python applications and modules. gettext defines a “translations” class which implements the parsing of GNU ‘.mo’ format files, and has methods for returning either standard 8-bit strings or Unicode strings. Translations instances can also install themselves in the built-in namespace as the function _().

```python
find (domain[, localedir[, languages]])
```

This function implements the standard ‘.mo’ file search algorithm. It takes a domain, identical to what textdomain() takes, and optionally a localedir (as in bindtextdomain()), and a list of languages. All arguments are strings.

If localedir is not given, then the default system locale directory is used. If languages is not given, then the following environment variables are searched: LANGUAGE, LC_ALL, LC_MESSAGES, and LANG. The first one returning a non-empty value is used for the languages variable. The environment variables can contain a colon separated list of languages, which will be split.

find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

‘localedir/language/LC_MESSAGES/domain.mo’

The first such file name that exists is returned by find(). If no such file is found, then None is returned.

```python
translation (domain[, localedir[, languages[, class_]]])
```

Return a Translations instance based on the domain, localedir, and languages, which are first passed to find() to get the associated ‘.mo’ file path. Instances with identical ‘.mo’ file names are cached. The actual class instantiated is either class_ if provided, otherwise GNUTranslations. The class’s constructor must take a single file object argument. If no ‘.mo’ file is found, this function raises IOError.

```python
install (domain[, localedir[, unicode]])
```

This installs the function _ in Python’s builtin namespace, based on domain, and localedir which are passed to the function translation(). The unicode flag is passed to the resulting translation object’s install method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the function _, e.g.

```python
print _(‘This string will be translated.’)
```

For convenience, you want the _() function to be installed in Python’s builtin namespace, so it is easily accessible in all modules of your application.

The NullTranslations class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is NullTranslations; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of NullTranslations:

```python
__init__ ([fp])
```

Takes an optional file object fp, which is ignored by the base class. Initializes “protected” instance variables _info and _charset which are set by derived classes. It then calls self._parse(fp) if fp is not None.

```python
_parse (fp)
```

No-op’d in the base class, this method takes file object fp, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

---

3See the footnote for bindtextdomain() above.
gettext (message)
Return the translated message. Overridden in derived classes.

ugettext (message)
Return the translated message as a Unicode string. Overridden in derived classes.

info ()
Return the “protected” _info variable.

charset ()
Return the “protected” _charset variable.

install ([unicode ])
If the unicode flag is false, this method installs self.gettext () into the built-in namespace, binding it to ‘_’. If unicode is true, it binds self.ugettext () instead. By default, unicode is false.

Note that this is only one way, albeit the most convenient way, to make the _ function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install _. Instead, they should use this code to make _ available to their module:

import gettext
t = gettext.translation(‘mymodule’, …)
_= t.gettext

This puts _ only in the module’s global namespace and so only affects calls within this module.

The GNUTranslations class

The gettext module provides one additional class derived from NullTranslations: GNUTranslations. This class overrides _parse () to enable reading GNU gettext format ‘.mo’ files in both big-endian and little-endian format.

It also parses optional meta-data out of the translation catalog. It is convention with GNU gettext to include meta-data as the translation for the empty string. This meta-data is in RFC 822-style key: value pairs. If the key Content-Type is found, then the charset property is used to initialize the “protected” _ charset instance variable. The entire set of key/value pairs are placed into a dictionary and set as the “protected” _ info instance variable.

If the ‘.mo’ file’s magic number is invalid, or if other problems occur while reading the file, instantiating a GNUTranslations class can raise IOError.

The other usefully overridden method is ugettext (), which returns a Unicode string by passing both the translated message string and the value of the “protected” _ charset variable to the builtin unicode () function.

Solaris message catalog support

The Solaris operating system defines its own binary ‘.mo’ file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the gettext module by James Henstridge, but this version has a slightly different API. Its documented usage was:
import gettext

cat = gettext.Catalog(domain, localedir)
_ = catgettext
print _('hello world')

For compatibility with this older module, the function Catalog() is an alias for the translation() function described above.
One difference between this module and Henstridge’s: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

6.25.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits.

In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the gettext module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in _('...') – i.e. a call to the function _(). For example:

```python
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

In this example, the string ‘writing a log message’ is marked as a candidate for translation, while the strings ‘mylog.txt’ and ‘w’ are not.

The Python distribution comes with two tools which help you generate the message catalogs once you’ve prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the ‘Tools/i18n’ directory.

The pygettext program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU gettext program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don’t need GNU gettext unless you’re also going to be translating C code (e.g. C extension modules).

pygettext generates textual Uniform-style human readable message catalog ‘.pot’ files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. pygettext is a command line script that supports a similar command line interface as xgettext; for details on its use, run:

François Pinard has written a program called xpot which does a similar job. It is available as part of his po-utils package at http://www.iro.umontreal.ca/contrib/po-utils/HTML.
Copies of these `.pot` files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a `.po` file. Using the `msgfmt.py` program (in the `Tools/i18n` directory), you take the `.po` files from your translators and generate the machine-readable `.mo` binary catalog files. The `.mo` files are what the `gettext` module uses for the actual translation processing during run-time.

How you use the `gettext` module in your code depends on whether you are internationalizing your entire application or a single module.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU `gettext` API but instead the class-based API.

Let’s say your module is called “spam” and the module's various natural language translation `.mo` files reside in `/usr/share/locale` in GNU `gettext` format. Here’s what you would put at the top of your module:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_= t.gettext
```

If your translators were providing you with Unicode strings in their `.po` files, you’d instead do:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_= t.ugettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_(`‘...’`) without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```python
import gettext
ggettext.install('myapplication')
```

If you need to set the locale directory or the `unicode` flag, you can pass these into the `install()` function:

```python
import gettext
ggettext.install('myapplication', '/usr/share/locale', unicode=1)
```

---

5 `msgfmt.py` is binary compatible with GNU `msgfmt` except that it provides a simpler, all-Python implementation. With this and `pygettext.py`, you generally won’t need to install the GNU `gettext` package to internationalize your Python applications.
Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```python
import gettext

lang1 = gettext.translation(languages=['en'])
lang2 = gettext.translation(languages=['fr'])
lang3 = gettext.translation(languages=['de'])

# start by using language 1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated were they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```python
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python',
           ]

# ...
for a in animals:
    print a
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:
def _(message): return message

animals = [_('mollusk'),
          _('albatross'),
          _('rat'),
          _('penguin'),
          _('python'),
]

del _

# ...
for a in animals:
    print _(a)

This works because the dummy definition of _() simply returns the string unchanged. And this dummy definition
will temporarily override any definition of _() in the built-in namespace (until the del command). Take care, though
if you have a previous definition of _ in the local namespace.

Note that the second use of _() will not identify “a” as being translatable to the pygettext program, since it is not a
string.

Another way to handle this is with the following example:

def N_(message): return message

animals = [N_('mollusk'),
          N_('albatross'),
          N_('rat'),
          N_('penguin'),
          N_('python'),
]

# ...
for a in animals:
    print _(a)

In this case, you are marking translatable strings with the function N_(), which won’t conflict with any definition of
_(). However, you will need to teach your message extraction program to look for translatable strings marked with
N_. pygettext and xpot both support this through the use of command line switches.

6.25.4 Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experi-
ence to the creation of this module:

- Peter Funk
- James Henstridge
- Marc-André Lemburg
- Martin von Löwis

6 The choice of N_() here is totally arbitrary; it could have just as easily been MarkThisStringForTranslation().
Optional Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modeled after the Unix or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here’s an overview:

- **signal**: Set handlers for asynchronous events.
- **socket**: Low-level networking interface.
- **select**: Wait for I/O completion on multiple streams.
- **thread**: Create multiple threads of control within one interpreter.
- **threading**: Higher-level threading interface.
- **Queue**: A synchronized queue class.
- **mmap**: Interface to memory-mapped files for Unix and Windows.
- **anydbm**: Generic interface to DBM-style database modules.
- **dumbdbm**: Portable implementation of the simple DBM interface.
- **dbhash**: DBM-style interface to the BSD database library.
- **whichdb**: Guess which DBM-style module created a given database.
- **bsddb**: Interface to Berkeley DB database library.
- **zlib**: Low-level interface to compression and decompression routines compatible with gzip.
- **gzip**: Interfaces for gzip compression and decompression using file objects.
- **zipfile**: Read and write ZIP-format archive files.
- **readline**: GNU readline support for Python.
- **rlcompleter**: Python identifier completion for the GNU readline library.

### 7.1 signal — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals and their handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (i.e. Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for SIGCHLD, which follows the underlying implementation.

- There is no way to “block” signals temporarily from critical sections (since this is not supported by all Unix flavors).

- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the “atomic” instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (e.g. regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.

- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after
the signal handler returns. This is dependent on the underlying UNIX system’s semantics regarding interrupted
system calls.

- Because the C signal handler always returns, it makes little sense to catch synchronous errors like SIGFPE or
  SIGSEGV.
- Python installs a small number of signal handlers by default: SIGPIPE is ignored (so write errors on pipes and
  sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a
  KeyboardInterrupt exception. All of these can be overridden.
- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to
  remember in using signals and threads simultaneously is: always perform signal() operations in the main
  thread of execution. Any thread can perform an alarm(), getsignal(), or pause(); only the main thread
  can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the
  Python signal module, even if the underlying thread implementation supports sending signals to individual
  threads). This means that signals can’t be used as a means of inter-thread communication. Use locks instead.

The variables defined in the signal module are:

**SIG_DFL**
This is one of two standard signal handling options; it will simply perform the default function for the signal.
For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action
for SIGCLD is to simply ignore it.

**SIG_IGN**
This is another standard signal handler, which will simply ignore the given signal.

**SIG***
All the signal numbers are defined symbolically. For example, the hangup signal is defined as
signal.SIGHUP; the variable names are identical to the names used in C programs, as found in
<signal.h>. The UNIX man page for 'signal()' lists the existing signals (on some systems this is sig-
nal(2), on others the list is in signal(7)). Note that not all systems define the same set of signal names; only
those names defined by the system are defined by this module.

**NSIG**
One more than the number of the highest signal number.

The signal module defines the following functions:

```python
alarm(time)
```
If time is non-zero, this function requests that a SIGALRM signal be sent to the process in time seconds. Any
previously scheduled alarm is canceled (i.e. only one alarm can be scheduled at any time). The returned value
is then the number of seconds before any previously set alarm was to have been delivered. If time is zero, no
alarm id scheduled, and any scheduled alarm is canceled. The return value is the number of seconds remaining
before a previously scheduled alarm. If the return value is zero, no alarm is currently scheduled. (See the UNIX
man page alarm(2).)

```python
getsignal(signalnum)
```
Return the current signal handler for the signal signalnum. The returned value may be a callable Python object,
or one of the special values signal.SIG_IGN, signal.SIG_DFL or None. Here, signal.SIG_IGN
means that the signal was previously ignored, signal.SIG_DFL means that the default way of handling the
signal was previously in use, and None means that the previous signal handler was not installed from Python.

```python
pause()
```
Causes the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.
(See the UNIX man page signal(2).)

```python
signal(signalnum, handler)
```
Set the handler for signal signalnum to the function handler. handler can be a callable Python object taking
two arguments (see below), or one of the special values signal.SIG_IGN or signal.SIG_DFL. The
previous signal handler will be returned (see the description of getsignal() above). (See the UNIX man page signal(2).)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a ValueError exception to be raised.

The handler is called with two arguments: the signal number and the current stack frame (None or a frame object; see the reference manual for a description of frame objects).

7.1.1 Example

Here is a minimal example program. It uses the alarm() function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the os.open() to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```python
import signal, os, FCNTL

def handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError, "Couldn't open device!"

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', FCNTL.O_RDWR)

signal.alarm(0)  # Disable the alarm
```

7.2 socket — Low-level networking interface

This module provides access to the BSD socket interface. It is available on all modern UNIX systems, Windows, MacOS, BeOS, OS/2, and probably additional platforms.

For an introduction to socket programming (in C), see the following papers: An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest and An Advanced 4.3BSD Interprocess Communication Tutorial, by Samuel J. Leffler et al, both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For UNIX, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification.

The Python interface is a straightforward transliteration of the UNIX system call and library interface for sockets to Python's object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Socket addresses are represented as a single string for the AF_UNIX address family and as a pair (host, port) for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like 'daring.cwi.nl' or an IP address like '100.50.200.5', and port is an integral port number. Other address families are currently not supported. The address format required by a particular socket object is automatically selected.
based on the address family specified when the socket object was created.

For IP addresses, two special forms are accepted instead of a host address: the empty string represents \texttt{INADDR\_ANY}, and the string '\texttt{<broadcast>}' represents \texttt{INADDR\_BROADCAST}.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error \texttt{socket.error}.

Non-blocking mode is supported through the \texttt{setblocking()} method.

The module \texttt{socket} exports the following constants and functions:

**exception error**

This exception is raised for socket- or address-related errors. The accompanying value is either a string telling what went wrong or a pair (\texttt{errno, string}) representing an error returned by a system call, similar to the value accompanying \texttt{os.error}. See the module \texttt{errno}, which contains names for the error codes defined by the underlying operating system.

\texttt{AF\_UNIX}
\texttt{AF\_INET}

These constants represent the address (and protocol) families, used for the first argument to \texttt{socket()}. If the \texttt{AF\_UNIX} constant is not defined then this protocol is unsupported.

\texttt{SOCK\_STREAM}
\texttt{SOCK\_DGRAM}
\texttt{SOCK\_RAW}
\texttt{SOCK\_RDM}
\texttt{SOCK\_SEQPACKET}

These constants represent the socket types, used for the second argument to \texttt{socket()}. (Only \texttt{SOCK\_STREAM} and \texttt{SOCK\_DGRAM} appear to be generally useful.)

\texttt{SO\_*}
\texttt{SOMAXCONN}
\texttt{MSG\_*}
\texttt{SOL\_*}
\texttt{IPPROTO\_*}
\texttt{IPPORT\_*}
\texttt{INADDR\_*}
\texttt{IP\_*}

Many constants of these forms, documented in the UNIX documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the \texttt{setsockopt()} and \texttt{getsockopt()} methods of socket objects. In most cases, only those symbols that are defined in the UNIX header files are defined; for a few symbols, default values are provided.

\texttt{getfqdn([name])}

Return a fully qualified domain name for \texttt{name}. If \texttt{name} is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by \texttt{gethostbyaddr()} is checked, then aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname is returned. New in version 2.0.

\texttt{gethostbyname(hostname)}

Translate a host name to IP address format. The IP address is returned as a string, e.g., '\texttt{100.50.200.5}' If the host name is an IP address itself it is returned unchanged. See \texttt{gethostbyname\_ex()} for a more complete interface.

\texttt{gethostbyname\_ex(hostname)}

Translate a host name to IP address format, extended interface. Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given \texttt{ip\_address}, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IP addresses for the same interface on the same host (often but not always a single address).
gethostname()
Return a string containing the hostname of the machine where the Python interpreter is currently executing. If you want to know the current machine’s IP address, use gethostbyname(gethostname()).

Note: gethostname() doesn’t always return the fully qualified domain name; use gethostbyaddr(gethostname()) (see below).

gethostbyaddr(ip_address)
Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IP addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function getfqdn().

getprotobyname(protocolname)
Translate an Internet protocol name (e.g. ‘icmp’) to a constant suitable for passing as the (optional) third argument to the socket() function. This is usually only needed for sockets opened in “raw” mode (SOCK_RAW); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

getservbyname(servicename, protocolname)
Translate an Internet service name and protocol name to a port number for that service. The protocol name should be ‘tcp’ or ‘udp’.

socket(family, type[, proto])
Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET or AF_UNIX. The socket type should be SOCK_STREAM, SOCK_DGRAM or perhaps one of the other ‘SOCK_’ constants. The protocol number is usually zero and may be omitted in that case.

fromfd(fd, family, type[, proto])
Build a socket object from an existing file descriptor (an integer as returned by a file object’s fileno() method). Address family, socket type and protocol number are as for the socket() function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (e.g. a server started by the UNIX inet daemon).

ntohl(x)
Convert 32-bit integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

ntohs(x)
Convert 16-bit integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

htonl(x)
Convert 32-bit integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

htons(x)
Convert 16-bit integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

inet_aton(ip_string)
Convert an IP address from dotted-quad string format (e.g. ‘123.45.67.89’) to 32-bit packed binary format, as a string four characters in length.

Useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary this function returns.

If the IP address string passed to this function is invalid, socket.error will be raised. Note that exactly what is valid depends on the underlying C implementation of inet_aton().

inet_ntoa(packed_ip)
Convert a 32-bit packed IP address (a string four characters in length) to its standard dotted-quad string repre-
sentation (e.g. '123.45.67.89').

Useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary this function takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, `socket.error` will be raised.

**SocketType**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

See Also:

**Module SocketServer** (section 11.14):

Classes that simplify writing network servers.

### 7.2.1 Socket Objects

Socket objects have the following methods. Except for `makefile()` these correspond to UNIX system calls applicable to sockets.

- **accept()**
  Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

- **bind(address)**
  Bind the socket to `address`. The socket must not already be bound. (The format of `address` depends on the address family — see above.) **Note:** This method has historically accepted a pair of parameters for `AF_INET` addresses instead of only a tuple. This was never intentional and is no longer be available in Python 2.0.

- **close()**
  Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

- **connect(address)**
  Connect to a remote socket at `address`. (The format of `address` depends on the address family — see above.) **Note:** This method has historically accepted a pair of parameters for `AF_INET` addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

- **connect_ex(address)**
  Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as “host not found,” can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the `errno` variable. This is useful, e.g., for asynchronous connects. **Note:** This method has historically accepted a pair of parameters for `AF_INET` addresses instead of only a tuple. This was never intentional and is no longer be available in Python 2.0 and later.

- **fileno()**
  Return the socket’s file descriptor (a small integer). This is useful with `select.select()`.

- **getpeername()**
  Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IP socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

- **getsockname()**
  Return the socket’s own address. This is useful to find out the port number of an IP socket, for instance. (The format of the address returned depends on the address family — see above.)

- **getsockopt(level, optname[, buflen])**
  Return the value of the given socket option (see the UNIX man page `getsockopt(2)`). The needed symbolic
constants (SO_* etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module struct for a way to decode C structures encoded as strings).

**listen** (backlog)

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

**makefile** ([mode], bufsize])

Return a file object associated with the socket. (File objects are described in 2.1.7, “File Objects.”) The file object references a dup()ed version of the socket file descriptor, so the file object and socket object may be closed or garbage-collected independently. The optional mode and bufsize arguments are interpreted the same way as by the built-in open() function.

**recv** (bufsize[, flags])

Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the UNIX manual page recv(2) for the meaning of the optional argument flags; it defaults to zero.

**recvfrom** (bufsize[, flags])

Receive data from the socket. The return value is a pair (string, address) where string is a string representing the data received and address is the address of the socket sending the data. The optional flags argument has the same meaning as for recv() above. (The format of address depends on the address family — see above.)

**send** (string[, flags])

Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Returns the number of bytes sent.

**sendto** (string[, flags], address)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address. The optional flags argument has the same meaning as for recv() above. Return the number of bytes sent. (The format of address depends on the address family — see above.)

**setblocking** (flag)

Set blocking or non-blocking mode of the socket: if flag is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a recv() call doesn’t find any data, or if a send() call can’t immediately dispose of the data, a error exception is raised; in blocking mode, the calls block until they can proceed.

**setsockopt** (level, optname, value)

Set the value of the given socket option (see the UNIX manual page setsockopt(2)). The needed symbolic constants are defined in the socket module (SO_* etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module struct for a way to encode C structures as strings).

**shutdown** (how)

Shut down one or both halves of the connection. If how is 0, further receives are disallowed. If how is 1, further sends are disallowed. If how is 2, further sends and receives are disallowed.

Note that there are no methods read() or write(); use recv() and send() without flags argument instead.

### 7.2.2 Example

Here are two minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence socket(), bind(), listen(), accept() (possibly repeating the accept() to service more than one client), while a client only needs the sequence socket(), connect(). Also note that the server does not send()/recv() on the socket it
is listening on but on the new socket returned by accept().

```python
# Echo server program
import socket

HOST = ''    # Symbolic name meaning the local host
PORT = 50007 # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007   # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

7.3 select — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on UNIX, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following:

**exception error**

The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

**poll()**

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section 7.3.1 below for the methods supported by polling objects.

**select (rfd, wfd, errfd[, timeout])**

This is a straightforward interface to the UNIX `select()` system call. The first three arguments are lists of ‘waitable objects’: either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer. The three lists of waitable objects are for input, output and ‘exceptional conditions’, respectively. Empty lists are allowed, but acceptance of three empty lists is platform-dependent. (It is known to work on UNIX but not on Windows.) The optional `timeout` argument specifies a time-out as a
floating point number in seconds. When the timeout argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Amongst the acceptable object types in the lists are Python file objects (e.g. sys.stdin, or objects returned by open() or os.popen()), socket objects returned by socket.socket(). You may also define a wrapper class yourself, as long as it has an appropriate fileno() method (that really returns a file descriptor, not just a random integer). Note: File objects on Windows are not acceptable, but sockets are. On Windows, the underlying select() function is provided by the WinSock library, and does not handle file descriptors that don’t originate from WinSock.

7.3.1 Polling Objects

The poll() system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. poll() scales better because the system call only requires listing the file descriptors of interest, while select() builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. select() is O(highest file descriptor), while poll() is O(number of file descriptors).

register(fd[, eventmask])

Register a file descriptor with the polling object. Future calls to the poll() method will then check whether the file descriptor has any pending I/O events. fd can be either an integer, or an object with a fileno() method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants POLLIN, POLLPRI, and POLLOUT, described in the table below. If not specified, the default value used will check for all 3 types of events.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>There is data to read</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>There is urgent data to read</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>Ready for output: writing will not block</td>
</tr>
<tr>
<td>POLLERR</td>
<td>Error condition of some sort</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>Hung up</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>Invalid request: descriptor not open</td>
</tr>
</tbody>
</table>

Registering a file descriptor that’s already registered is not an error, and has the same effect as registering the descriptor exactly once.

unregister(fd)

Remove a file descriptor being tracked by a polling object. Just like the register() method, fd can be an integer or an object with a fileno() method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a KeyError exception to be raised.

poll([timeout])

Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. fd is the file descriptor, and event is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report.

7.4 thread — Multiple threads of control
This module provides low-level primitives for working with multiple threads (a.k.a. light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (a.k.a. mutexes or binary semaphores) are provided.

The module is optional. It is supported on Windows NT and '95, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. “pthread”) implementation.

It defines the following constant and functions:

**exception error**

Raised on thread-specific errors.

**LockType**

This is the type of lock objects.

**start_new_thread** *(function, args[, kwargs]*)

Start a new thread. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

**exit** ()

Raise the *SystemExit* exception. When not caught, this will cause the thread to exit silently.

**exit_thread** ()

Deprecated since release 1.5.2. Use exit(). This is an obsolete synonym for exit().

**allocate_lock** ()

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

**get_ident** ()

Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

Lock objects have the following methods:

**acquire** ([*waitflag]*)

Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that’s their reason for existence), and returns *None*. If the integer *waitflag* argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. If an argument is present, the return value is 1 if the lock is acquired successfully, 0 if not.

**release** ()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

**locked** ()

Return the status of the lock: 1 if it has been acquired by some thread, 0 if not.

**Caveats:**

- Threads interact strangely with interrupts: the *KeyboardInterrupt* exception will be received by an arbitrary thread. (When the *signal* module is available, interrupts always go to the main thread.)
- Calling *sys.exit()* or raising the *SystemExit* exception is equivalent to calling *exit()*.
- Not all built-in functions that may block waiting for I/O allow other threads to run. (The most popular ones (*time.sleep()*), *file.read()*), *select.select()* work as expected.)
• It is not possible to interrupt the acquire() method on a lock — the KeyboardInterrupt exception will happen after the lock has been acquired.

• When the main thread exits, it is system defined whether the other threads survive. On SGI IRIX using the native thread implementation, they survive. On most other systems, they are killed without executing try ... finally clauses or executing object destructors.

• When the main thread exits, it does not do any of its usual cleanup (except that try ... finally clauses are honored), and the standard I/O files are not flushed.

7.5 threading — Higher-level threading interface

This module constructs higher-level threading interfaces on top of the lower level thread module. This module is safe for use with ‘from threading import *’. It defines the following functions and objects:

activeCount()
Return the number of currently active Thread objects. The returned count is equal to the length of the list returned by enumerate(). A function that returns the number of currently active threads.

Condition()
A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

currentThread()
Return the current Thread object, corresponding to the caller’s thread of control. If the caller’s thread of control was not created through the threading module, a dummy thread object with limited functionality is returned.

enumerate()
Return a list of all currently active Thread objects. The list includes daemonic threads, dummy thread objects created by currentThread(), and the main thread. It excludes terminated threads and threads that have not yet been started.

Event()
A factory function that returns a new event object. An event manages a flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

Lock()
A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

RLock()
A factory function that returns a new reentrant lock object. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Semaphore()
A factory function that returns a new semaphore object. A semaphore manages a counter representing the number of release() calls minus the number of acquire() calls, plus an initial value. The acquire() method blocks if necessary until it can return without making the counter negative.

class Thread()
A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

Detailed interfaces for the objects are documented below.

The design of this module is loosely based on Java’s threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python’s Thread class supports a subset
of the behavior of Java’s Thread class; currently, there are no priorities, no thread groups, and threads cannot be
destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java’s Thread class, when implemented,
are mapped to module-level functions.

All of the methods described below are executed atomically.

### 7.5.1 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is
currently the lowest level synchronization primitive available, implemented directly by the thread extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic
methods, `acquire()` and `release()`. When the state is unlocked, `acquire()` changes the state to locked and
returns immediately. When the state is locked, `acquire()` blocks until a call to `release()` in another thread
changes it to unlocked, then the `acquire()` call resets it to locked and returns. The `release()` method should
only be called in the locked state; it changes the state to unlocked and returns immediately. When more than one thread
is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()`
call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across
implementations.

All methods are executed atomically.

```python
acquire([blocking = 1])
Acquire a lock, blocking or non-blocking.
When invoked without arguments, block until the lock is unlocked, then set it to locked, and return. There is no
return value in this case.
When invoked with the blocking argument set to true, do the same thing as when called without arguments, and
return true.
When invoked with the blocking argument set to false, do not block. If a call without an argument would block,
return false immediately; otherwise, do the same thing as when called without arguments, and return true.
```

```python
release()
Release a lock.
When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to
become unlocked, allow exactly one of them to proceed.
Do not call this method when the lock is unlocked.
There is no return value.
```

### 7.5.2 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it
uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive
locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its `acquire()` method; this returns once the thread owns the lock. To unlock the
lock, a thread calls its `release()` method. `acquire() / release()` call pairs may be nested; only the final
`release()` (i.e. the `release()` of the outermost pair) resets the lock to unlocked and allows another thread
blocked in `acquire()` to proceed.

```python
acquire([blocking = 1])
Acquire a lock, blocking or non-blocking.
When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and
return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock
is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more
than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the `blocking` argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the `blocking` argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

`release()`
Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. Do not call this method when the lock is unlocked. There is no return value.

### 7.5.3 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. (Passing one in is useful when several condition variables must share the same lock.)

A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock. It also has a `wait()` method, and `notify()` and `notifyAll()` methods. These three must only be called when the calling thread has acquired the lock.

The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notifyAll()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notifyAll()` methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notifyAll()` finally relinquishes ownership of the lock.

Tip: the typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notifyAll()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```python
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

To choose between `notify()` and `notifyAll()`, consider whether one state change can be interesting for only
one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

```python
class Condition(lock):
    If the lock argument is given and not None, it must be a Lock or RLock object, and it is used as the underlying lock. Otherwise, a new RLock object is created and used as the underlying lock.

acquire(*args)
    Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()
    Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait([timeout])
    Wait until notified or until a timeout occurs. This must only be called when the calling thread has acquired the lock.
    This method releases the underlying lock, and then blocks until it is awakened by a notify() or notifyAll() call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.
    When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).
    When the underlying lock is an RLock, it is not released using its release() method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the RLock class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

notify()
    Wake up a thread waiting on this condition, if any. This must only be called when the calling thread has acquired the lock.
    This method wakes up one of the threads waiting for the condition variable, if any are waiting; it is a no-op if no threads are waiting.
    The current implementation wakes up exactly one thread, if any are waiting. However, it’s not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than one thread.
    Note: the awakened thread does not actually return from its wait() call until it can reacquire the lock. Since notify() does not release the lock, its caller should.

notifyAll()
    Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one.
```

### 7.5.4 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used P() and V() instead of acquire() and release()).

A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

```python
class Semaphore([value])
    The optional argument gives the initial value for the internal counter; it defaults to 1.

acquire([blocking])
    Acquire a semaphore.
```
When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release() to make it larger than zero. This is done with proper interlocking so that if multiple acquire() calls are blocked, release() will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with blocking set to true, do the same thing as when called without arguments, and return true.

When invoked with blocking set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

**release()**
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

### 7.5.5 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and one or more other threads are waiting for it.

An event object manages an internal flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

**class Event()**
The internal flag is initially false.

**isSet()**
Return true if and only if the internal flag is true.

**set()**
Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call wait() once the flag is true will not block at all.

**clear()**
Reset the internal flag to false. Subsequently, threads calling wait() will block until set() is called to set the internal flag to true again.

**wait([timeout])**
Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls set() to set the flag to true, or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

### 7.5.6 Thread Objects

This class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the __init__() and run() methods of this class.

Once a thread object is created, its activity must be started by calling the thread’s start() method. This invokes the run() method in a separate thread of control.

Once the thread’s activity is started, the thread is considered ‘alive’ and ‘active’ (these concepts are almost, but not quite exactly, the same; their definition is intentionally somewhat vague). It stops being alive and active when its run() method terminates – either normally, or by raising an unhandled exception. The isAlive() method tests whether the thread is alive.
Other threads can call a thread’s `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, set with the `setName()` method, and retrieved with the `getName()` method.

A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set with the `setDaemon()` method and retrieved with the `getDaemon()` method.

There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”. These are threads of control started outside the threading module, e.g. directly from C code. Dummy thread objects have limited functionality; they are always considered alive, active, and daemonic, and cannot be `join()`ed. They are never deleted, since it is impossible to detect the termination of alien threads.

```python
class Thread (group=None, target=None, name=None, args=(), kwargs=–˝)
    This constructor should always be called with keyword arguments. Arguments are:
    group Should be None; reserved for future extension when a ThreadGroup class is implemented.
    target Callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.
    name The thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small
    decimal number.
    args Argument tuple for the target invocation. Defaults to ()
    kwargs Keyword argument dictionary for the target invocation. Defaults to {}.
    If the subclass overrides the constructor, it must make sure to invoke the base class constructor
    (Thread.__init__()) before doing anything else to the thread.

start()
    Start the thread’s activity.
    This must be called at most once per thread object. It arranges for the object’s run() method to be invoked in
    a separate thread of control.

run()
    Method representing the thread’s activity.
    You may override this method in a subclass. The standard run() method invokes the callable object passed to
    the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the
    args and kwargs arguments, respectively.

join([timeout])
    Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called
    terminates – either normally or through an unhandled exception – or until the optional timeout occurs.
    When the timeout argument is present and not None, it should be a floating point number specifying a timeout
    for the operation in seconds (or fractions thereof).
    A thread can be join()ed many times.
    A thread cannot join itself because this would cause a deadlock.
    It is an error to attempt to join() a thread before it has been started.

getName()
    Return the thread’s name.

setName(name)
    Set the thread’s name.
    The name is a string used for identification purposes only. It has no semantics. Multiple threads may be given
    the same name. The initial name is set by the constructor.
```
isAlive()
Return whether the thread is alive.
Roughly, a thread is alive from the moment the start() method returns until its run() method terminates.

isDaemon()
Return the thread’s daemon flag.

setDaemon(daemonic)
Set the thread’s daemon flag to the Boolean value daemonic. This must be called before start() is called.
The initial value is inherited from the creating thread.
The entire Python program exits when no active non-daemon threads are left.

7.6 Queue — A synchronized queue class

The Queue module implements a multi-producer, multi-consumer FIFO queue. It is especially useful in threads programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. It depends on the availability of thread support in Python.

The Queue module defines the following class and exception:

class Queue(maxsize)
Constructor for the class. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

exception Empty
Exception raised when non-blocking get() (or get_nowait()) is called on a Queue object which is empty or locked.

exception Full
Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full or locked.

7.6.1 Queue Objects

Class Queue implements queue objects and has the methods described below. This class can be derived from in order to implement other queue organizations (e.g. stack) but the inheritable interface is not described here. See the source code for details. The public methods are:

qsize()
Return the approximate size of the queue. Because of multithreading semantics, this number is not reliable.

empty()
Return 1 if the queue is empty, 0 otherwise. Because of multithreading semantics, this is not reliable.

full()
Return 1 if the queue is full, 0 otherwise. Because of multithreading semantics, this is not reliable.

put(item[, block])
Put item into the queue. If optional argument block is 1 (the default), block if necessary until a free slot is available. Otherwise (block is 0), put item on the queue if a free slot is immediately available, else raise the Full exception.

put_nowait(item)
Equivalent to put(item, 0).
get ([block])
Remove and return an item from the queue. If optional argument block is 1 (the default), block if necessary until an item is available. Otherwise (block is 0), return an item if one is immediately available, else raise the Empty exception.

get_nowait ()
Equivalent to get (0).

7.7 mmap — Memory-mapped file support

Memory-mapped file objects behave like both mutable strings and like file objects. You can use mmap objects in most places where strings are expected; for example, you can use the re module to search through a memory-mapped file. Since they’re mutable, you can change a single character by doing obj[index] = ‘a’, or change a substring by assigning to a slice: obj[i1:i2] = ‘...’. You can also read and write data starting at the current file position, and seek () through the file to different positions.

A memory-mapped file is created by the following function, which is different on Unix and on Windows.

mmap (fileno, length[, tagname])
(Windows version) Maps length bytes from the file specified by the file handle fileno, and returns a mmap object. If length is 0, the maximum length of the map will be the current size of the file when mmap () is called.

If you wish to map an existing Python file object, use its fileno () method to obtain the correct value for the fileno parameter. The file must be opened for update.

tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between UNIX and Windows.

mmap (fileno, size[, flags, prot])
_UNIX version) Maps length bytes from the file specified by the file handle fileno, and returns a mmap object. If you wish to map an existing Python file object, use its fileno () method to obtain the correct value for the fileno parameter. The file must be opened for update.

flags specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that’s shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. prot defaults to PROT_READ | PROT_WRITE.

Memory-mapped file objects support the following methods:

close ()
Close the file. Subsequent calls to other methods of the object will result in an exception being raised.

find (string[, start])
Returns the lowest index in the object where the substring string is found. Returns -1 on failure. start is the index at which the search begins, and defaults to zero.

flush ([offset, size])
Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If offset and size are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed.

move (dest, src, count)
Copy the count bytes starting at offset src to the destination index dest.
read(num)
Return a string containing up to num bytes starting from the current file position; the file position is updated to point after the bytes that were returned.

read_byte()
Returns a string of length 1 containing the character at the current file position, and advances the file position by 1.

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(newsize)

seek(pos[, whence])
Set the file’s current position. whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end).

size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()
Returns the current position of the file pointer.

write(string)
Write the bytes in string into memory at the current position of the file pointer; the file position is updated to point after the bytes that were written.

write_byte(byte)
Write the single-character string byte into memory at the current position of the file pointer; the file position is advanced by 1.

7.8 anydbm — Generic access to DBM-style databases

anydbm is a generic interface to variants of the DBM database — dbhash (requires bsddb), gdbm, or dbm. If none of these modules is installed, the slow-but-simple implementation in module dumbdbm will be used.

open (filename[, flag[, mode]])
Open the database file filename and return a corresponding object.

The optional flag argument can be ‘r’ to open an existing database for reading only, ‘w’ to open an existing database for reading and writing, ‘c’ to create the database if it doesn’t exist, or ‘n’, which will always create a new empty database. If not specified, the default value is ‘r’.

The optional mode argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

exception error
A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception anydbm.error as the first item — the latter is used when anydbm.error is raised.

The object returned by open() supports most of the same functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the has_key() and keys() methods are available. Keys and values must always be strings.

See Also:
Module anydbm (section 7.8):
Generic interface to dbm-style databases.
Module dbhash (section 7.10):
BSD db database interface.

Module dbm (section 8.6):
Standard UNIX database interface.

Module dumbdbm (section 7.9):
Portable implementation of the dbm interface.

Module gdbm (section 8.7):
GNU database interface, based on the dbm interface.

Module shelve (section 3.17):
General object persistence built on top of the Python dbm interface.

Module whichdb (section 7.11):
Utility module used to determine the type of an existing database.

7.9 dumbdbm — Portable DBM implementation

A simple and slow database implemented entirely in Python. This should only be used when no other DBM-style database is available.

open (filename[, flag[, mode]])
Open the database file filename and return a corresponding object. The optional flag argument can be ‘r’ to open an existing database for reading only, ‘w’ to open an existing database for reading and writing, ‘c’ to create the database if it doesn’t exist, or ‘n’, which will always create a new empty database. If not specified, the default value is ‘r’.

The optional mode argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

exception error
Raised for errors not reported as KeyError errors.

See Also:
Module anydbm (section 7.8):
Generic interface to dbm-style databases.

Module whichdb (section 7.11):
Utility module used to determine the type of an existing database.

7.10 dbhash — DBM-style interface to the BSD database library

The dbhash module provides a function to open databases using the BSD db library. This module mirrors the interface of the other Python database modules that provide access to DBM-style databases. The bsddb module is required to use dbhash.

This module provides an exception and a function:

exception error
Exception raised on database errors other than KeyError. It is a synonym for bsddb.error.

open (path, flag[, mode])
Open a db database and return the database object. The path argument is the name of the database file.

The flag argument can be ‘r’ (the default), ‘w’, ‘c’ (which creates the database if it doesn’t exist), or ‘n’ (which always creates a new empty database). For platforms on which the BSD db library supports locking, an
’1’ can be appended to indicate that locking should be used.

The optional mode parameter is used to indicate the UNIX permission bits that should be set if a new database must be created; this will be masked by the current umask value for the process.

See Also:
Module anydbm (section 7.8):
    Generic interface to dbm-style databases.
Module bsddb (section 7.12):
    Lower-level interface to the BSD db library.
Module whichdb (section 7.11):
    Utility module used to determine the type of an existing database.

7.10.1 Database Objects

The database objects returned by open() provide the methods common to all the DBM-style databases. The following methods are available in addition to the standard methods.

first()
    It’s possible to loop over every key in the database using this method and the next() method. The traversal is ordered by the databases internal hash values, and won’t be sorted by the key values. This method returns the starting key.

last()
    Return the last key in a database traversal. This may be used to begin a reverse-order traversal; see previous().

next(key)
    Returns the key that follows key in the traversal. The following code prints every key in the database db, without having to create a list in memory that contains them all:
    
    ```python
    k = db.first()
    while k != None:
        print k
        k = db.next(k)
    ```

previous(key)
    Return the key that comes before key in a forward-traversal of the database. In conjunction with last(), this may be used to implement a reverse-order traversal.

sync()
    This method forces any unwritten data to be written to the disk.

7.11 whichdb — Guess which DBM module created a database

The single function in this module attempts to guess which of the several simple database modules available—dbm, gdbm, or dbhash—should be used to open a given file.

whichdb(filename)
    Returns one of the following values: None if the file can’t be opened because it’s unreadable or doesn’t exist; the empty string (‘’) if the file’s format can’t be guessed; or a string containing the required module name, such as ‘dbm’ or ‘gdbm’.
The **bsddb** module provides an interface to the Berkeley DB library. Users can create hash, btree or record based library files using the appropriate open call. Bsddb objects behave generally like dictionaries. Keys and values must be strings, however, so to use other objects as keys or to store other kinds of objects the user must serialize them somehow, typically using marshal.dumps or pickle.dumps.

There are two incompatible versions of the underlying library. Version 1.85 is widely available, but has some known bugs. Version 2 is not quite as widely used, but does offer some improvements. The **bsddb** module uses the 1.85 interface. Starting with Python 2.0, the **configure** script can usually determine the version of the library which is available and build it correctly. If you have difficulty getting **configure** to do the right thing, run it with the **--help** option to get information about additional options that can help. On Windows, you will need to define the **HAVE_DB_185_H** macro if you are building Python from source and using version 2 of the DB library.

The **bsddb** module defines the following functions that create objects that access the appropriate type of Berkeley DB file. The first two arguments of each function are the same. For ease of portability, only the first two arguments should be used in most instances.

```python
hashopen(filename[, flag[, mode[, bsize[, ffactor[, nelem[, cachesize[, hash[, lorder ]]]]]]]])
```

Open the hash format file named `filename`. The optional `flag` identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.

```python
btopen(filename[, flag[, mode[, btflags[, cachesize[, maxkeypage[, minkeypage[, psize[, lorder ]]]]]]]])
```

Open the btree format file named `filename`. The optional `flag` identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.

```python
rnopen(filename[, flag[, mode[, rnflags[, cachesize[, psize[, lorder[, reclen[, bval[, bfname ]]]]]]]]])
```

Open a DB record format file named `filename`. The optional `flag` identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.

See Also:

- **Module dbhash** (section 7.10): DBM-style interface to the **bsddb**

### 7.12.1 Hash, BTree and Record Objects

Once instantiated, hash, btree and record objects support the following methods:

**close()**

Close the underlying file. The object can no longer be accessed. Since there is no open open method for these objects, to open the file again a new **bsddb** module open function must be called.

**keys()**

Return the list of keys contained in the DB file. The order of the list is unspecified and should not be relied on. In particular, the order of the list returned is different for different file formats.

**has_key(key)**

Return 1 if the DB file contains the argument as a key.

**set_location(key)**

Set the cursor to the item indicated by `key` and return a tuple containing the key and its value. For binary tree
databases (opened using `btopen()`), if `key` does not actually exist in the database, the cursor will point to the
next item in sorted order and return that key and value. For other databases, `KeyError` will be raised if `key` is
not found in the database.

`first()`

Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified, except in
the case of B-Tree databases.

`next()`

Set the cursor to the next item in the DB file and return it. The order of keys in the file is unspecified, except in
the case of B-Tree databases.

`previous()`

Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified, except in
the case of B-Tree databases. This is not supported on hashtable databases (those opened with `hashopen()`).

`last()`

Set the cursor to the last item in the DB file and return it. The order of keys in the file is unspecified. This is not
supported on hashtable databases (those opened with `hashopen()`).

`synchronize`()

Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified. This is not
supported on hashtable databases (those opened with `hashopen()`).

Example:

```python
>>> import bsddb
>>> db = bsddb.btopen('/tmp/spam.db', 'c')
>>> for i in range(10): db['%d' % i] = '%d' % (i * i)
...   ...   ...
>>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> db.sync()
0
```

7.13 `zlib` — Compression compatible with `gzip`

For applications that require data compression, the functions in this module allow compression and decompression,
using the zlib library. The zlib library has its own home page at `http://www.info-zip.org/pub/infozip/zlib/`. Version 1.1.3 is
the most recent version as of September 2000; use a later version if one is available. There are known incompatibilities
between the Python module and earlier versions of the zlib library.

The available exception and functions in this module are:

`exception error`
Exception raised on compression and decompression errors.

**adler32** ([string], value)

Computes a Adler-32 checksum of string. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If value is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several input strings. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures.

**compress** ([string], level)

Compresses the data in string, returning a string contained compressed data. level is an integer from 1 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. The default value is 6. Raises the error exception if any error occurs.

**compressobj** ([level])

Returns a compression object, to be used for compressing data streams that won’t fit into memory at once. level is an integer from 1 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. The default value is 6.

**crc32** ([string], value)

Computes a CRC (Cyclic Redundancy Check) checksum of string. If value is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several input strings. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures.

**decompress** ([string], wbits, bufsize)

Decompresses the data in string, returning a string containing the uncompressed data. The wbits parameter controls the size of the window buffer. If bufsize is given, it is used as the initial size of the output buffer. Raises the error exception if any error occurs.

The absolute value of wbits is the base two logarithm of the size of the history buffer (the “window size”) used when compressing data. Its absolute value should be between 8 and 15 for the most recent versions of the zlib library, larger values resulting in better compression at the expense of greater memory usage. The default value is 15. When wbits is negative, the standard gzip header is suppressed; this is an undocumented feature of the zlib library, used for compatibility with unzip's compression file format.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don’t have to get this value exactly right; tuning it will only save a few calls to malloc(). The default size is 16384.

**decompressobj** ([wbits])

Returns a compression object, to be used for decompressing data streams that won’t fit into memory at once. The wbits parameter controls the size of the window buffer.

Compression objects support the following methods:

**compress** (string)

Compress string, returning a string containing compressed data for at least part of the data in string. This data should be concatenated to the output produced by any preceding calls to the compress() method. Some input may be kept in internal buffers for later processing.

**flush** ([mode])

All pending input is processed, and a string containing the remaining compressed output is returned. mode can be selected from the constants Z_SYNC_FLUSH, Z_FULL_FLUSH, or Z_FINISH, defaulting to Z_FINISH. Z_SYNC_FLUSH and Z_FULL_FLUSH allow compressing further strings of data and are used to allow partial error recovery on decompression, while Z_FINISH finishes the compressed stream and prevents compressing any more data. After calling flush() with mode set to Z_FINISH, the compress() method cannot be called again; the only realistic action is to delete the object.

Decompression objects support the following methods, and a single attribute:
**unused_data**

A string which contains any unused data from the last string fed to this decompression object. If the whole string turned out to contain compressed data, this is ",", the empty string.

The only way to determine where a string of compressed data ends is by actually decompressing it. This means that when compressed data is contained part of a larger file, you can only find the end of it by reading data and feeding it into a decompression object's `decompress` method until the `unused_data` attribute is no longer the empty string.

**decompress**(string)

Decompress string, returning a string containing the uncompressed data corresponding to at least part of the data in `string`. This data should be concatenated to the output produced by any preceding calls to the `decompress()` method. Some of the input data may be preserved in internal buffers for later processing.

**flush()**

All pending input is processed, and a string containing the remaining uncompressed output is returned. After calling `flush()`, the `decompress()` method cannot be called again; the only realistic action is to delete the object.

See Also:

Module `gzip` (section 7.14):

Reading and writing `gzip`-format files.

http://www.info-zip.org/pub/infozip/zlib/

The zlib library home page.

7.14 gzip — Support for gzip files

The data compression provided by the `zlib` module is compatible with that used by the GNU compression program `gzip`. Accordingly, the `gzip` module provides the GzipFile class to read and write `gzip`-format files, automatically compressing or decompressing the data so it looks like an ordinary file object. Note that additional file formats which can be decompressed by the `gzip` and `gunzip` programs, such as those produced by `compress` and `pack`, are not supported by this module.

The module defines the following items:

**class GzipFile([[filename[, mode[, compresslevel[, fileobj ]]]]])**

Constructor for the GzipFile class, which simulates most of the methods of a file object, with the exception of the `seek()` and `tell()` methods. At least one of `fileobj` and `filename` must be given a non-trivial value.

The new class instance is based on `fileobj`, which can be a regular file, a `StringIO` object, or any other object which simulates a file. It defaults to `None`, in which case `filename` is opened to provide a file object.

When `fileobj` is not `None`, the `filename` argument is only used to be included in the `gzip` file header, which may includes the original filename of the uncompressed file. It defaults to the filename of `fileobj`, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The `mode` argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, or `'wb'`, depending on whether the file will be read or written. The default is the mode of `fileobj` if discernible; otherwise, the default is `'rb'`. Be aware that only the `'rb'`, `'ab'`, and `'wb'` values should be used for cross-platform portability.

The `compresslevel` argument is an integer from 1 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. The default is 9.

Calling a GzipFile object's `close()` method does not close `fileobj`, since you might wish to append more material after the compressed data. This also allows you to pass a `StringIO` object opened for writing as `fileobj`, and retrieve the resulting memory buffer using the `StringIO` object's `getvalue()` method.

**open([filename[, mode[, compresslevel ]]])**

This is a shorthand for GzipFile(`filename`, `mode`, `compresslevel`). The `filename` argument is required;
mode defaults to ‘rb’ and compresslevel defaults to 9.

See Also:

Module zlib (section 7.13):
The basic data compression module needed to support the gzip file format.

7.15 zipfile — Work with ZIP archives

New in version 1.6.

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle ZIP files which have appended comments, or multi-disk ZIP files.

The available attributes of this module are:

exception error
The error raised for bad ZIP files.

class ZipFile (...)
The class for reading and writing ZIP files. See “ZipFile Objects” (section 7.15.1) for constructor details.

class PyZipFile (...)
Class for creating ZIP archives containing Python libraries.

class ZipInfo ([filename[, date_time ] ])
Class used to represent information about a member of an archive. Instances of this class are returned by the getinfo() and infolist() methods of ZipFile objects. Most users of the zipfile module will not need to create these, but only use those created by this module. filename should be the full name of the archive member, and date_time should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section 7.15.3, “ZipInfo Objects.”

is_zipfile (filename)
Returns true if filename is a valid ZIP file based on its magic number, otherwise returns false. This module does not currently handle ZIP files which have appended comments.

ZIP_STORED
The numeric constant for an uncompressed archive member.

ZIP_DEFLATED
The numeric constant for the usual ZIP compression method. This requires the zlib module. No other compression methods are currently supported.

See Also:

PKZIP Application Note
(http://www.pkware.com/appnote.html)
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page
(http://www.info-zip.org/pub/infozip/)
Information about the Info-ZIP project’s ZIP archive programs and development libraries.

7.15.1 ZipFile Objects

class ZipFile ([filename[, mode[, compression ] ]])
Open a ZIP file named filename. The mode parameter should be ’r’ to read an existing file, ’w’ to truncate
and write a new file, or ‘a’ to append to an existing file. For mode is ‘a’ and filename refers to an existing
ZIP file, then additional files are added to it. If filename does not refer to a ZIP file, then a new ZIP archive is
appended to the file. This is meant for adding a ZIP archive to another file, such as ‘python.exe’. Using

    cat myzip.zip >> python.exe

also works, and at least WinZip can read such files. compression is the ZIP compression method to use
when writing the archive, and should be ZIP_STORED or ZIP_DEFLATED; unrecognized values will cause
RuntimeError to be raised. If ZIP_DEFLATED is specified but the zlib module is not available, Runtime
Error is also raised. The default is ZIP_STORED.

close()
    Close the archive file. You must call close() before exiting your program or essential records will not be
written.

getinfo(name)
    Return a ZipInfo object with information about the archive member name.

infolist()
    Return a list containing a ZipInfo object for each member of the archive. The objects are in the same order
as their entries in the actual ZIP file on disk if an existing archive was opened.

namelist()
    Return a list of archive members by name.

printdir()
    Print a table of contents for the archive to sys.stdout.

read(name)
    Return the bytes of the file in the archive. The archive must be open for read or append.

testzip()
    Read all the files in the archive and check their CRC’s. Return the name of the first bad file, or else return None.

write(filename, arcname, compress_type]
    Write the file named filename to the archive, giving it the archive name arcname (by default, this will be the same
as filename). If given, compress_type overrides the value given for the compression parameter to the constructor
for the new entry. The archive must be open with mode ‘w’ or ‘a’.

writestr(zinfo, bytes)
    Write the string bytes to the archive; meta-information is given as the ZipInfo instance zinfo. At least the
filename, date, and time must be given by zinfo. The archive must be opened with mode ‘w’ or ‘a’.

The following data attribute is also available:

depend
    The level of debug output to use. This may be set from 0 (the default, no output) to 3 (the most output).
    Debugging information is written to sys.stdout.

7.15.2 PyZipFile Objects

The PyZipFile constructor takes the same parameters as the ZipFile constructor. Instances have one method in
addition to those of ZipFile objects.

writepy(pathname, basename]
    Search for files ‘*.py’ and add the corresponding file to the archive. The corresponding file is a ‘*.pyo’ file if
available, else a ‘*.pyc’ file, compiling if necessary. If the pathname is a file, the filename must end with ‘.py’,
and just the (corresponding ‘*.py[co]’) file is added at the top level (no path information). If it is a directory, and
the directory is not a package directory, then all the files ‘*.py[co]’ are added at the top level. If the directory is a
package directory, then all ‘*.py[oc]’ are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively. basename is intended for internal use only. The writepy() method makes archives with file names like this:

```
string.pyc # Top level name
test/__init__.pyc # Package directory
test/testall.pyc # Module test.testall
test/bogus/__init__.pyc # Subpackage directory
test/bogus/myfile.pyc # Submodule test.bogus.myfile
```

7.15.3 ZipInfo Objects

Instances of the ZipInfo class are returned by the getinfo() and infolist() methods of ZipFile objects. Each object stores information about a single member of the ZIP archive.

Instances have the following attributes:

**filename**
Name of the file in the archive.

**date_time**
The time and date of the last modification to to the archive member. This is a tuple of six values:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Year</td>
</tr>
<tr>
<td>1</td>
<td>Month (one-based)</td>
</tr>
<tr>
<td>2</td>
<td>Day of month (one-based)</td>
</tr>
<tr>
<td>3</td>
<td>Hours (zero-based)</td>
</tr>
<tr>
<td>4</td>
<td>Minutes (zero-based)</td>
</tr>
<tr>
<td>5</td>
<td>Seconds (zero-based)</td>
</tr>
</tbody>
</table>

**compress_type**
Type of compression for the archive member.

**comment**
Comment for the individual archive member.

**extra**
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this string.

**create_system**
System which created ZIP archive.

**create_version**
PKZIP version which created ZIP archive.

**extract_version**
PKZIP version needed to extract archive.

**reserved**
Must be zero.

**flag_bits**
ZIP flag bits.

**volume**
Volume number of file header.

**internal_attr**
Internal attributes.

**external_attr**
External file attributes.

**header_offset**
Byte offset to the file header.

**file_offset**
Byte offset to the start of the file data.

**CRC**
CRC-32 of the uncompressed file.

**compress_size**
Size of the compressed data.

**file_size**
Size of the uncompressed file.

## 7.16 readline — GNU readline interface

The `readline` module defines a number of functions used either directly or from the `rlcompleter` module to facilitate completion and history file read and write from the Python interpreter.

The `readline` module defines the following functions:

- **`parse_and_bind(string)`**
  Parse and execute single line of a readline init file.

- **`get_line_buffer()`**
  Return the current contents of the line buffer.

- **`insert_text(string)`**
  Insert text into the command line.

- **`read_init_file([filename])`**
  Parse a readline initialization file. The default filename is the last filename used.

- **`read_history_file([filename])`**
  Load a readline history file. The default filename is ‘’/.history’.

- **`write_history_file([filename])`**
  Save a readline history file. The default filename is ‘’/.history’.

- **`get_history_length()`**
  Return the desired length of the history file. Negative values imply unlimited history file size.

- **`set_history_length(length)`**
  Set the number of lines to save in the history file. `write_history_file()` uses this value to truncate the history file when saving. Negative values imply unlimited history file size.

- **`set_completer([function])`**
  Set or remove the completer function. The completer function is called as `function(text, state), for i in [0, 1, 2, ...]` until it returns a non-string. It should return the next possible completion starting with `text`.

- **`get_begidx()`**
  Get the beginning index of the readline tab-completion scope.

- **`get_endidx()`**
  Get the ending index of the readline tab-completion scope.
**set_completer_delims**(string)

Set the readline word delimiters for tab-completion.

**get_completer_delims**()

Get the readline word delimiters for tab-completion.

**See Also:**

Module **rlcompleter** (section 7.17):
Completion of Python identifiers at the interactive prompt.

### 7.16.1 Example

The following example demonstrates how to use the `readline` module’s history reading and writing functions to automatically load and save a history file named `.pyhist` from the user’s home directory. The code below would normally be executed automatically during interactive sessions from the user’s PYTHONSTARTUP file.

```python
import os
histfile = os.path.join(os.environ["HOME"], ".pyhist")
try:
    readline.read_history_file(histfile)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

### 7.17 rlcompleter — Completion function for GNU readline

The `rlcompleter` module defines a completion function for the `readline` module by completing valid Python identifiers and keywords.

This module is UNIX-specific due to it’s dependence on the `readline` module.

The `rlcompleter` module defines the `Completer` class.

Example:

```python
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline.<TAB PRESSED>
readline.__doc__     readline.get_line_buffer     readline.read_init_file
readline.__file__    readline.insert_text        readline.set_completer
readline.__name__    readline.parse_and_bind
>>> readline.
```

The `rlcompleter` module is designed for use with Python’s interactive mode. A user can add the following lines to his or her initialization file (identified by the PYTHONSTARTUP environment variable) to get automatic Tab completion:
try:
    import readline
except ImportError:
    print "Module readline not available."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")

7.17.1 Completer Objects

Completer objects have the following method:

**complete**(text, state)

Return the state th completion for text.

If called for text that doesn’t include a period character (‘.’), it will complete from names currently defined in __main__, __builtin__ and keywords (as defined by the keyword module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (i.e., functions will not be evaluated, but it can generate calls to __getattr__() upto the last part, and find matches for the rest via the dir() function.
Unix Specific Services

The modules described in this chapter provide interfaces to features that are unique to the UNIX operating system, or in some cases to some or many variants of it. Here’s an overview:

- **posix**: The most common POSIX system calls (normally used via module os).
- **pwd**: The password database (getpwnam() and friends).
- **grp**: The group database (getgrnam() and friends).
- **crypt**: The crypt() function used to check UNIX passwords.
- **dl**: Call C functions in shared objects.
- **dbm**: The standard “database” interface, based on ndbm.
- **gdbm**: GNU’s reinterpretation of dbm.
- **termios**: POSIX style tty control.
- **TERMIOS**: Symbolic constants required to use the termios module.
- **tty**: Utility functions that perform common terminal control operations.
- **pty**: Pseudo-Terminal Handling for SGI and Linux.
- **fcntl**: The fcntl() and ioctl() system calls.
- **pipes**: A Python interface to UNIX shell pipelines.
- **posixfile**: A file-like object with support for locking.
- **resource**: An interface to provide resource usage information on the current process.
- **nis**: Interface to Sun’s NIS (a.k.a. Yellow Pages) library.
- **syslog**: An interface to the UNIX syslog library routines.
- **commands**: Utility functions for running external commands.

### 8.1 posix — The most common POSIX system calls

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised UNIX interface).

**Do not import this module directly.** Instead, import the module os, which provides a portable version of this interface. On UNIX, the os module provides a superset of the posix interface. On non-UNIX operating systems the posix module is not available, but a subset is always available through the os interface. Once os is imported, there is no performance penalty in using it instead of posix. In addition, os provides some additional functionality, such as automatically calling putenv() when an entry in os.environ is changed.

The descriptions below are very terse; refer to the corresponding UNIX manual (or POSIX documentation) entry for more information. Arguments called path refer to a pathname given as a string.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise error (a synonym for the standard exception OSError), described below.
8.1.1 Large File Support

Several operating systems (including AIX, HPUX, Irix and Solaris) provide support for files that are larger than 2 Gb from a C programming model where \texttt{int} and \texttt{long} are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as \textit{large files}.

Large file support is enabled in Python when the size of an \texttt{off_t} is larger than a \texttt{long} and the \texttt{long long} type is available and is at least as large as an \texttt{off_t}. Python longs are then used to represent file sizes, offsets and other values that can exceed the range of a Python int. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

\begin{verbatim}
CFLAGS='\texttt{getconf LFS_CFLAGS}' OPT='\texttt{-g -O2} $\texttt{CFLAGS}' \\
./configure
\end{verbatim}

On large-file-capable Linux systems, this might work:

\begin{verbatim}
CC='\texttt{-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64}'
export CC
./configure
\end{verbatim}

8.1.2 Module Contents

Module \texttt{posix} defines the following data item:

\texttt{environ}

A dictionary representing the string environment at the time the interpreter was started. For example, \texttt{environ['HOME']} is the path name of your home directory, equivalent to \texttt{getenv("HOME")} in C.

Modifying this dictionary does not affect the string environment passed on by \texttt{execv()}, \texttt{popen()} or \texttt{system()}; if you need to change the environment, pass \texttt{environ} to \texttt{execve()} or add variable assignments and export statements to the command string for \texttt{system()} or \texttt{popen()}.

\textbf{Note:} The \texttt{os} module provides an alternate implementation of \texttt{environ} which updates the environment on modification. Note also that updating \texttt{os.environ} will render this dictionary obsolete. Use of the \texttt{os} for this is recommended over direct access to the \texttt{posix} module.

Additional contents of this module should only be accessed via the \texttt{os} module; refer to the documentation for that module for further information.

8.2 \texttt{pwd} — The password database

This module provides access to the UNIX user account and password database. It is available on all UNIX versions.

Password database entries are reported as 7-tuples containing the following items from the password database (see \texttt{<pwd.h>}), in order:
The uid and gid items are integers, all others are strings. `KeyError` is raised if the entry asked for cannot be found.

**Note:** In traditional UNIX the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm (see module `crypt`). However, most modern unices use a so-called shadow password system. On those unices the field `pw_passwd` only contains an asterisk (`'*'`) or the letter `x` where the encrypted password is stored in a file `'/etc/shadow'` which is not world readable.

It defines the following items:

- `getpwuid (uid)`
  Return the password database entry for the given numeric user ID.

- `getpwnam (name)`
  Return the password database entry for the given user name.

- `getpwall()`
  Return a list of all available password database entries, in arbitrary order.

**See Also:**

- Module `grp` (section 8.3):
  An interface to the group database, similar to this.

### 8.3  `grp` — The group database

This module provides access to the UNIX group database. It is available on all UNIX versions.

Group database entries are reported as 4-tuples containing the following items from the group database (see `<grp.h>`), in order:

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gr_name</td>
<td>the name of the group</td>
</tr>
<tr>
<td>1</td>
<td>gr_passwd</td>
<td>the (encrypted) group password; often empty</td>
</tr>
<tr>
<td>2</td>
<td>gr_gid</td>
<td>the numerical group ID</td>
</tr>
<tr>
<td>3</td>
<td>gr_mem</td>
<td>all the group member's user names</td>
</tr>
</tbody>
</table>

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information.)

It defines the following items:

- `getggrgid (gid)`
  Return the group database entry for the given numeric group ID. `KeyError` is raised if the entry asked for cannot be found.

- `getgrnam (name)`
  Return the group database entry for the given group name. `KeyError` is raised if the entry asked for cannot be found.
getgrall()
Return a list of all available group entries, in arbitrary order.

See Also:
Module `pwd` (section 8.2):
An interface to the user database, similar to this.

8.4 crypt — Function to check UNIX passwords

This module implements an interface to the `crypt(3)` routine, which is a one-way hash function based upon a modified DES algorithm; see the UNIX man page for further details. Possible uses include allowing Python scripts to accept typed passwords from the user, or attempting to crack UNIX passwords with a dictionary.

crypt(word, salt)
word will usually be a user’s password as typed at a prompt or in a graphical interface. salt is usually a random two-character string which will be used to perturb the DES algorithm in one of 4096 ways. The characters in salt must be in the set `[./a-zA-Z0-9]`. Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt (the first two characters represent the salt itself).

A simple example illustrating typical use:

```python
import crypt, getpass, pwd

def login():
    username = raw_input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise "Sorry, currently no support for shadow passwords"
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptedpasswd[:2]) == cryptedpasswd
    else:
        return 1
```

8.5 dl — Call C functions in shared objects

The dl module defines an interface to the `dlopen()` function, which is the most common interface on UNIX platforms for handling dynamically linked libraries. It allows the program to call arbitrary functions in such a library.

Note: This module will not work unless

```
    sizeof(int) == sizeof(long) == sizeof(char *)
```

If this is not the case, SystemError will be raised on import.

The dl module defines the following function:

dlopen(name[, mode = RTLD_LAZY])
Open a shared object file, and return a handle. Mode signifies late binding (RTLD_LAZY) or immediate binding (RTLD_NOW). Default is RTLD_LAZY. Note that some systems do not support RTLD_NOW.

Return value is a dlobject.
The `dl` module defines the following constants:

- **RTLD_LAZY**  
  Useful as an argument to `open()`.  

- **RTLD_NOW**  
  Useful as an argument to `open()`. Note that on systems which do not support immediate binding, this constant will not appear in the module. For maximum portability, use `hasattr()` to determine if the system supports immediate binding.

The `dl` module defines the following exception:

- **exception error**  
  Exception raised when an error has occurred inside the dynamic loading and linking routines.

Example:

```python
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

This example was tried on a Debian GNU/Linux system, and is a good example of the fact that using this module is usually a bad alternative.

### 8.5.1 DI Objects

DI objects, as returned by `open()` above, have the following methods:

- **close()**  
  Free all resources, except the memory.

- **sym(name)**  
  Return the pointer for the function named `name`, as a number, if it exists in the referenced shared object, otherwise `None`. This is useful in code like:

  ```python
  >>> if a.sym('time'):
  ...     a.call('time')
  ... else:
  ...     time.time()
  ```

  (Note that this function will return a non-zero number, as zero is the NULL pointer)

- **call(name, arg1[, arg2... ])**
  Call the function named `name` in the referenced shared object. The arguments must be either Python integers, which will be passed as is, Python strings, to which a pointer will be passed, or `None`, which will be passed as NULL. Note that strings should only be passed to functions as `const char*`, as Python will not like its string mutated.

  There must be at most 10 arguments, and arguments not given will be treated as `None`. The function’s return value must be a C `long`, which is a Python integer.

### 8.6 dbm — Simple “database” interface

The `dbm` module provides an interface to the UNIX (`n)dbm` library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn’t print the keys and values, and the `items()` and `values()` methods are not supported.
This module can be used with the “classic” ndbm interface, the BSD DB compatibility interface, or the GNU GDBM compatibility interface. On UNIX, the configure script will attempt to locate the appropriate header file to simplify building this module.

The module defines the following:

**exception error**
Raised on dbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

**library**
Name of the ndbm implementation library used.

**open (filename[, flag[, mode ]])**
Open a dbm database and return a dbm object. The filename argument is the name of the database file (without the `.dir` or `.pag` extensions; note that the BSD DB implementation of the interface will append the extension `.db` and only create one file).

The optional flag argument must be one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'r'</code></td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td><code>'w'</code></td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td><code>'c'</code></td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td><code>'n'</code></td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The optional mode argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal 0666.

See Also:

Module anydbm (section 7.8):
Generic interface to dbm-style databases.

Module gdbm (section 8.7):
Similar interface to the GNU GDBM library.

Module whichdb (section 7.11):
Utility module used to determine the type of an existing database.

### 8.7 gdbm — GNU’s reinterpretation of dbm

This module is quite similar to the dbm module, but uses gdbm instead to provide some additional functionality. Please note that the file formats created by gdbm and dbm are incompatible.

The gdbm module provides an interface to the GNU DBM library. gdbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a gdbm object doesn’t print the keys and values, and the items() and values() methods are not supported.

The module defines the following constant and functions:

**exception error**
Raised on gdbm-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

**open (filename[, flag[, mode ]])**
Open a gdbm database and return a gdbm object. The filename argument is the name of the database file.

The optional flag argument can be `'r'` (to open an existing database for reading only — default), `'w'` (to open an existing database for reading and writing), `'c'` (which creates the database if it doesn’t exist), or `'n'` (which always creates a new empty database).
The following additional characters may be appended to the flag to control how the database is opened:

- `'f'` — Open the database in fast mode. Writes to the database will not be synchronized.
- `'s'` — Synchronized mode. This will cause changes to the database will be immediately written to the file.
- `'u'` — Do not lock database.

Not all flags are valid for all versions of `gdbm`. The module constant `open_flags` is a string of supported flag characters. The exception `error` is raised if an invalid flag is specified.

The optional `mode` argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal 0666.

In addition to the dictionary-like methods, `gdbm` objects have the following methods:

**firstkey()**

It’s possible to loop over every key in the database using this method and the `nextkey()` method. The traversal is ordered by `gdbm`’s internal hash values, and won’t be sorted by the key values. This method returns the starting key.

**nextkey(key)**

Returns the key that follows `key` in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```python
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

**reorganize()**

If you have carried out a lot of deletions and would like to shrink the space used by the `gdbm` file, this routine will reorganize the database. `gdbm` will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

**sync()**

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

See Also:

Module `anydbm` (section 7.8):

Generic interface to `dbm`-style databases.

Module `whichdb` (section 7.11):

Utility module used to determine the type of an existing database.

### 8.8 termios — POSIX style tty control

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see the POSIX or UNIX manual pages. It is only available for those UNIX versions that support POSIX `termios` style tty I/O control (and only if configured at installation time).

All functions in this module take a file descriptor `fd` as their first argument. This must be an integer file descriptor, such as returned by `sys.stdin.fileno()`.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.
The module defines the following functions:

**tcgetattr**(fd)
Return a list containing the tty attributes for file descriptor fd, as follows: `[iflag, oflag, cflag, iflag, ispeed, ospeed, cc]` where cc is a list of the tty special characters (each a string of length 1, except the items with indices VMIN and VTIME, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the cc array must be done using the symbolic constants defined in the termios module.

**tcsetattr**(fd, when, attributes)
Set the tty attributes for file descriptor fd from the attributes, which is a list like the one returned by tcgetattr(). The when argument determines when the attributes are changed: TCSANOW to change immediately, TCSADRAIN to change after transmitting all queued output, or TCSAFLUSH to change after transmitting all queued output and discarding all queued input.

**tcsendbreak**(fd, duration)
Send a break on file descriptor fd. A zero duration sends a break for 0.25–0.5 seconds; a nonzero duration has a system dependent meaning.

**tcdrain**(fd)
Wait until all output written to file descriptor fd has been transmitted.

**tcflush**(fd, queue)
Discard queued data on file descriptor fd. The queue selector specifies which queue: TCIFLUSH for the input queue, TCOFLUSH for the output queue, or TCIOFLUSH for both queues.

**tcflow**(fd, action)
Suspend or resume input or output on file descriptor fd. The action argument can be TCOFF to suspend output, TCQON to restart output, TCIQFF to suspend input, or TCQION to restart input.

See Also:
Module tty (section 8.10):
Convenience functions for common terminal control operations.

### 8.8.1 Example

Here’s a function that prompts for a password with echoing turned off. Note the technique using a separate tcgetattr() call and a try ... finally statement to ensure that the old tty attributes are restored exactly no matter what happens:

```python
def getpass(prompt = "Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```
8.9 TERMIOS — Constants used with the termios module

**Deprecated since release 2.1.** Import needed constants from `termios` instead.

This module defines the symbolic constants required to use the `termios` module (see the previous section). See the POSIX or UNIX manual pages for a list of those constants.

8.10 tty — Terminal control functions

The `tty` module defines functions for putting the tty into cbreak and raw modes. Because it requires the `termios` module, it will work only on UNIX.

The `tty` module defines the following functions:

- `setraw(fd[, when])`
  Change the mode of the file descriptor `fd` to raw. If `when` is omitted, it defaults to `TERMIOS.TCAFLUSH`, and is passed to `termios.tcsetattr()`.

- `setcbreak(fd[, when])`
  Change the mode of file descriptor `fd` to cbreak. If `when` is omitted, it defaults to `TERMIOS.TCAFLUSH`, and is passed to `termios.tcsetattr()`.

See Also:
- Module `termios` (section 8.8):
  Low-level terminal control interface.
- Module `TERMIOS` (section 8.9):
  Constants useful for terminal control operations.

8.11 pty — Pseudo-terminal utilities

The `pty` module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically. Because pseudo-terminal handling is highly platform dependant, there is code to do it only for SGI and Linux. (The Linux code is supposed to work on other platforms, but hasn’t been tested yet.)

The `pty` module defines the following functions:

- `fork()`
  Fork. Connect the child’s controlling terminal to a pseudo-terminal. Return value is `(pid, fd)`. Note that the child gets `pid 0`, and the `fd` is invalid. The parent’s return value is the `pid` of the child, and `fd` is a file descriptor connected to the child’s controlling terminal (and also to the child’s standard input and output).

- `openpty()`
  Open a new pseudo-terminal pair, using `os.openpty()` if possible, or emulation code for SGI and generic UNIX systems. Return a pair of file descriptors `(master, slave)`, for the master and the slave end, respectively.

- `spawn(argv[, master_read[, stdin_read]])`
  Spawn a process, and connect its controlling terminal with the current process’s standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

  The functions `master_read` and `stdin_read` should be functions which read from a file-descriptor. The defaults try to read 1024 bytes each time they are called.
8.12  fcntl — The fcntl() and ioctl() system calls

This module performs file control and I/O control on file descriptors. It is an interface to the fcntl() and ioctl() UNIX routines. File descriptors can be obtained with the fileno() method of a file or socket object.

The module defines the following functions:

fcntl(fd, op[, arg])
Perform the requested operation on file descriptor fd. The operation is defined by op and is operating system dependent. Typically these codes can be retrieved from the library module FCNTL. The argument arg is optional, and defaults to the integer value 0. When present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the C fcntl() call. When the argument is a string it represents a binary structure, e.g. created by struct.pack(). The binary data is copied to a buffer whose address is passed to the C fcntl() call. The return value after a successful call is the contents of the buffer, converted to a string object. The length of the returned string will be the same as the length of the arg argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the fcntl() fails, an IOError is raised.

ioctl(fd, op, arg)
This function is identical to the fcntl() function, except that the operations are typically defined in the library module IOCTL.

flock(fd, op)
Perform the lock operation op on file descriptor fd. See the UNIX manual flock(3) for details. (On some systems, this function is emulated using fcntl().)

lockf(fd, operation, [len, [start, [whence]]])
This is essentially a wrapper around the fcntl() locking calls. fd is the file descriptor of the file to lock or unlock, and operation is one of the following values:

• LOCK_UN – unlock
• LOCK_SH – acquire a shared lock
• LOCK_EX – acquire an exclusive lock

When operation is LOCK_SH or LOCK_EX, it can also be bit-wise OR’d with LOCK_NB to avoid blocking on lock acquisition. If LOCK_NB is used and the lock cannot be acquired, an IOError will be raised and the exception will have an errno attribute set to EACCES or EAGAIN (depending on the operating system; for portability, check for both values).

length is the number of bytes to lock, start is the byte offset at which the lock starts, relative to whence, and whence is as with fileobj.seek(), specifically:

• 0 – relative to the start of the file (SEEK_SET)
• 1 – relative to the current buffer position (SEEK_CUR)
• 2 – relative to the end of the file (SEEK_END)

The default for start is 0, which means to start at the beginning of the file. The default for length is 0 which means to lock to the end of the file. The default for whence is also 0.

If the library modules FCNTL or IOCTL are missing, you can find the opcodes in the C include files <sys/fcntl.h> and <sys/ioctl.h>. You can create the modules yourself with the h2py script, found in the 'Tools/scripts/' directory.

Examples (all on a SVR4 compliant system):
import struct, fcntl, FCNTL

file = open(...)  # Open a file
rv = fcntl(file.fileno(), FCNTL.F_SETFL, FCNTL.O_NDELAY)

lockdata = struct.pack('hhllhh', FCNTL.F_WRLCK, 0, 0, 0, 0, 0)  # Create lock data
rv = fcntl.fcntl(file.fileno(), FCNTL.F_SETLKW, lockdata)

Note that in the first example the return value variable rv will hold an integer value; in the second example it will hold a string value. The structure lay-out for the lockdata variable is system dependent — therefore using the flock() call may be better.

### 8.13 pipes — Interface to shell pipelines

The `pipes` module defines a class to abstract the concept of a pipeline — a sequence of convertors from one file to another.

Because the module uses /bin/sh command lines, a POSIX or compatible shell for `os.system()` and `os.popen()` is required.

The `pipes` module defines the following class:

```python
class Template():  # An abstraction of a pipeline.
```

**Example:**

```python
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('/tmp/1', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('/tmp/1').read()
'HELLO WORLD'
```

### 8.13.1 Template Objects

Template objects following methods:

**reset()**
Restore a pipeline template to its initial state.

**clone()**
Return a new, equivalent, pipeline template.

**debug(flag)**
If flag is true, turn debugging on. Otherwise, turn debugging off. When debugging is on, commands to be executed are printed, and the shell is given set -x command to be more verbose.

**append(cmd, kind)**
Append a new action at the end. The cmd variable must be a valid bourne shell command. The kind variable consists of two letters.

The first letter can be either of ‘-‘ (which means the command reads its standard input), ‘f’ (which means
the commands reads a given file on the command line) or ‘.’ (which means the commands reads no input, and hence must be first.)

Similarly, the second letter can be either of ‘-’ (which means the command writes to standard output), ‘f’ (which means the command writes a file on the command line) or ‘.’ (which means the command does not write anything, and hence must be last.)

`prepend(cmd, kind)`
Add a new action at the beginning. See `append()` for explanations of the arguments.

`open(file, mode)`
Return a file-like object, open to `file`, but read from or written to by the pipeline. Note that only one of ‘r’, ‘w’ may be given.

`copy(infile, outfile)`
Copy `infile` to `outfile` through the pipe.

8.14 **posixfile — File-like objects with locking support**

**Note:** This module will become obsolete in a future release. The locking operation that it provides is done better and more portably by the `fcntl.lockf()` call.

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of UNIX, since it uses `fcntl.fcntl()` for file locking.

To instantiate a posixfile object, use the `open()` function in the `posixfile` module. The resulting object looks and feels roughly the same as a standard file object.

The `posixfile` module defines the following constants:

**SEEK_SET**
Offset is calculated from the start of the file.

**SEEK_CUR**
Offset is calculated from the current position in the file.

**SEEK_END**
Offset is calculated from the end of the file.

The `posixfile` module defines the following functions:

`open(filename[, mode[, bufsize]])`
Create a new posixfile object with the given filename and mode. The `filename`, `mode` and `bufsize` arguments are interpreted the same way as by the built-in `open()` function.

`fileopen(fileobject)`
Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

`lock(fmt[, len[, start[, whence]]])`
Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The `len` argument specifies the length of the section that should be locked. The default is 0. `start` specifies the starting offset of the section, where the default is 0. The `whence` argument specifies where the offset is relative to. It accepts one of the constants `SEEK_SET`, `SEEK_CUR` or `SEEK_END`. The default is `SEEK_SET`. For more information about the arguments refer to the `fcntl(2)` manual page on your system.

`flags([flags])`
Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old flags, unless specified otherwise. The format is explained below in a table. Without the flags argument a string indicating the current flags is returned (this is the same as the ‘?’ modifier). For more information about the flags refer to the fcntl(2) manual page on your system.

dup()
Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

dup2(fd)
Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

file()
Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

All methods raise IOError when the request fails.

Format characters for the lock() method have the following meaning:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘u’</td>
<td>unlock the specified region</td>
</tr>
<tr>
<td>‘r’</td>
<td>request a read lock for the specified section</td>
</tr>
<tr>
<td>‘w’</td>
<td>request a write lock for the specified section</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘!’</td>
<td>wait until the lock has been granted</td>
<td></td>
</tr>
<tr>
<td>‘?’</td>
<td>return the first lock conflicting with the requested lock, or None if there is no conflict.</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Note:

(1) The lock returned is in the format (mode, len, start, whence, pid) where mode is a character representing the type of lock (‘r’ or ‘w’). This modifier prevents a request from being granted; it is for query purposes only.

Format characters for the flags() method have the following meanings:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘a’</td>
<td>append only flag</td>
</tr>
<tr>
<td>‘c’</td>
<td>close on exec flag</td>
</tr>
<tr>
<td>‘n’</td>
<td>no delay flag (also called non-blocking flag)</td>
</tr>
<tr>
<td>‘s’</td>
<td>synchronization flag</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘!’</td>
<td>turn the specified flags ‘off’, instead of the default ‘on’</td>
<td>(1)</td>
</tr>
<tr>
<td>‘!’</td>
<td>replace the flags, instead of the default ’OR’ operation</td>
<td>(1)</td>
</tr>
<tr>
<td>‘?’</td>
<td>return a string in which the characters represent the flags that are set.</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

(1) The ‘!’ and ‘=’ modifiers are mutually exclusive.
(2) This string represents the flags after they may have been altered by the same call.

Examples:

```python
import posixfile

file = posixfile.open('/tmp/test', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

8.15 resource — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

A single exception is defined for errors:

```python
exception error
```

The functions described below may raise this error if the underlying system call failures unexpectedly.

8.15.1 Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit` man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

```python
getrlimit (resource)
```

Returns a tuple `(soft, hard)` with the current soft and hard limits of `resource`. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

```python
setrlimit (resource, limits)
```

Sets new limits of consumption of `resource`. The `limits` argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of -1 can be used to specify the maximum possible upper limit.

Raises `ValueError` if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit (unless the process has an effective UID of super-user). Can also raise `error` if the underlying system call fails.

These symbols define resources whose consumption can be controlled using the `setrlimit()` and `getrlimit()` functions described below. The values of these symbols are exactly the constants used by C programs.

The UNIX man page for `getrlimit(2)` lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource.

```python
RLIMIT_CORE
```

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of
a partial core file if a larger core would be required to contain the entire process image.

**RLIMIT_CPU**
The maximum amount of CPU time (in seconds) that a process can use. If this limit is exceeded, a **SIGXCPU** signal is sent to the process. (See the `signal` module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

**RLIMITFSIZE**
The maximum size of a file which the process may create. This only affects the stack of the main thread in a multi-threaded process.

**RLIMIT_DATA**
The maximum size (in bytes) of the process’s heap.

**RLIMIT_STACK**
The maximum size (in bytes) of the call stack for the current process.

**RLIMIT_RSS**
The maximum resident set size that should be made available to the process.

**RLIMIT_NPROC**
The maximum number of processes the current process may create.

**RLIMIT_NOFILE**
The maximum number of open file descriptors for the current process.

**RLIMIT_OFILE**
The BSD name for **RLIMIT_NOFILE**.

**RLIMIT_MEMLOC**
The maximum address space which may be locked in memory.

**RLIMIT_VMEM**
The largest area of mapped memory which the process may occupy.

**RLIMIT_AS**
The maximum area (in bytes) of address space which may be taken by the process.

### 8.15.2 Resource Usage

These functions are used to retrieve resource usage information:

**getrusage** *(who)*
This function returns a large tuple that describes the resources consumed by either the current process or its children, as specified by the `who` parameter. The `who` parameter should be specified using one of the `RUSAGE_*` constants described below.

The elements of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

The first two elements of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the `getrusage(2)` man page for detailed information about these values. A brief summary is presented here:
<table>
<thead>
<tr>
<th>Offset</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>time in user mode (float)</td>
</tr>
<tr>
<td>1</td>
<td>time in system mode (float)</td>
</tr>
<tr>
<td>2</td>
<td>maximum resident set size</td>
</tr>
<tr>
<td>3</td>
<td>shared memory size</td>
</tr>
<tr>
<td>4</td>
<td>unshared memory size</td>
</tr>
<tr>
<td>5</td>
<td>unshared stack size</td>
</tr>
<tr>
<td>6</td>
<td>page faults not requiring I/O</td>
</tr>
<tr>
<td>7</td>
<td>page faults requiring I/O</td>
</tr>
<tr>
<td>8</td>
<td>number of swap outs</td>
</tr>
<tr>
<td>9</td>
<td>block input operations</td>
</tr>
<tr>
<td>10</td>
<td>block output operations</td>
</tr>
<tr>
<td>11</td>
<td>messages sent</td>
</tr>
<tr>
<td>12</td>
<td>messages received</td>
</tr>
<tr>
<td>13</td>
<td>signals received</td>
</tr>
<tr>
<td>14</td>
<td>voluntary context switches</td>
</tr>
<tr>
<td>15</td>
<td>involuntary context switches</td>
</tr>
</tbody>
</table>

This function will raise a ValueError if an invalid who parameter is specified. It may also raise an error exception in unusual circumstances.

**getpagesize()**

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.) This function is useful for determining the number of bytes of memory a process is using. The third element of the tuple returned by getrusage() describes memory usage in pages; multiplying by page size produces number of bytes.

The following RUSAGE_* symbols are passed to the getrusage() function to specify which processes information should be provided for.

**RUSAGE_SELF**

RUSAGE_SELF should be used to request information pertaining only to the process itself.

**RUSAGE_CHILDREN**

Pass to getrusage() to request resource information for child processes of the calling process.

**RUSAGE_BOTH**

Pass to getrusage() to request resources consumed by both the current process and child processes. May not be available on all systems.

### 8.16 nis — Interface to Sun’s NIS (Yellow Pages)

The nis module gives a thin wrapper around the NIS library, useful for central administration of several hosts.

Because NIS exists only on UNIX systems, this module is only available for UNIX.

The nis module defines the following functions:

**match(key, mapname)**

Return the match for key in map mapname, or raise an error (nis.error) if there is none. Both should be strings, key is 8-bit clean. Return value is an arbitrary array of bytes (i.e., may contain NULL and other joys).

Note that mapname is first checked if it is an alias to another name.

**cat (mapname)**

Return a dictionary mapping key to value such that match (key, mapname) == value. Note that both keys and values of the dictionary are arbitrary arrays of bytes.

Note that mapname is first checked if it is an alias to another name.
maps()
Return a list of all valid maps.

The nis module defines the following exception:

exception error
An error raised when a NIS function returns an error code.

8.17 syslog — UNIX syslog library routines

This module provides an interface to the UNIX syslog library routines. Refer to the UNIX manual pages for a detailed description of the syslog facility.

The module defines the following functions:

syslog([priority,] message)
Send the string message to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a facility and a level. The optional priority argument, which defaults to LOG_INFO, determines the message priority. If the facility is not encoded in priority using logical-or (LOG_INFO | LOG_USER), the value given in the openlog() call is used.

openlog([ident[, logopt[, facility]]])
Logging options other than the defaults can be set by explicitly opening the log file with openlog() prior to calling syslog(). The defaults are (usually) ident = 'syslog', logopt = 0, facility = LOG_USER. The ident argument is a string which is prepended to every message. The optional logopt argument is a bit field - see below for possible values to combine. The optional facility argument sets the default facility for messages which do not have a facility explicitly encoded.

closelog()
Close the log file.

setlogmask(maskpri)
Set the priority mask to maskpri and return the previous mask value. Calls to syslog() with a priority level not set in maskpri are ignored. The default is to log all priorities. The function LOG_MASK(pri) calculates the mask for the individual priority pri. The function LOG_UPTO(pri) calculates the mask for all priorities up to and including pri.

The module defines the following constants:

Priority levels (high to low): LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG.

Facilities: LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, LOG_NEWS, LOG_UUCP, LOG_CRON and LOG_LOCAL0 to LOG_LOCAL7.

Log options: LOG_PID, LOG_CONS, LOG_NDELAY, LOG_NOWAIT and LOG_PERROR if defined in <syslog.h>.

8.18 commands — Utilities for running commands

The commands module contains wrapper functions for os.popen() which take a system command as a string and return any output generated by the command and, optionally, the exit status.

The commands module defines the following functions:
getstatusoutput (cmd)
Execute the string cmd in a shell with os.popen() and return a 2-tuple (status, output). cmd is actually run as (cmd ; )2>&1, so that the returned output will contain output or error messages. A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function wait().

getoutput (cmd)
Like getstatusoutput(), except the exit status is ignored and the return value is a string containing the command’s output.

getstatus (file)
Return the output of ‘ls -ld file’ as a string. This function uses the getoutput() function, and properly escapes backslashes and dollar signs in the argument.

Example:

```python
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x 1 root 13352 Oct 14 1994 /bin/ls'
```
The Python Debugger

The module `pdb` defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible — it is actually defined as the class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules `bdb` (undocumented) and `cmd`.

The debugger’s prompt is ‘`(Pdb)`’. Typical usage to run a program under control of the debugger is:

```python
>>> import pdb
>>> import mymodule
>>> pdb.run("mymodule.test()")
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

`pdb.py` can also be invoked as a script to debug other scripts. For example:

```
python /usr/local/lib/python1.5/pdb.py myscript.py
```

Typical usage to inspect a crashed program is:
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./mymodule.py", line 4, in test
test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)

The module defines the following functions; each enters the debugger in a slightly different way:

**run**(statement[, globals[, locals]])
Execute the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type ‘continue’, or you can step through the statement using ‘step’ or ‘next’ (all these commands are explained below). The optional globals and locals arguments specify the environment in which the code is executed; by default the dictionary of the module __main__ is used. (See the explanation of the exec statement or the eval() built-in function.)

**runeval**(expression[, globals[, locals]])
Evaluate the expression (given as a string) under debugger control. When runeval() returns, it returns the value of the expression. Otherwise this function is similar to run().

**runcall**(function[, argument, ...])
Call the function (a function or method object, not a string) with the given arguments. When runcall() returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

**set_trace()**
Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

**post_mortem**(traceback)
Enter post-mortem debugging of the given traceback object.

**pm()**
Enter post-mortem debugging of the traceback found in sys.last_traceback.

### 9.1 Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. ‘h(elp)’ means that either ‘h’ or ‘help’ can be used to enter the help command (but not ‘he’ or ‘hel’, nor ‘H’ or ‘Help’ or ‘HELP’). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets (‘[]’) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (‘|’).

Entering a blank line repeats the last command entered. Exception: if the last command was a ‘list’ command, the next 11 lines are listed.

Commands that the debugger doesn’t recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (‘!’). This is a
powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger’s state is not changed.

Multiple commands may be entered on a single line, separated by ’;;’. (A single ’;’ is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first ’;;’ pair, even if it is in the middle of a quoted string.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

If a file ‘.pdbrc’ exists in the user’s home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

h(elp) [command] Without argument, print the list of available commands. With a command as argument, print help about that command. ‘help pdb’ displays the full documentation file; if the environment variable PAGER is defined, the file is piped through that command instead. Since the command argument must be an identifier, ‘help exec’ must be entered to get help on the ‘!’ command.

w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

d(own) Move the current frame one level down in the stack trace (to a newer frame).

u(p) Move the current frame one level up in the stack trace (to an older frame).

b(reak) [[filename:]lineno [function[, condition ]]] With a lineno argument, set a break there in the current file. With a function argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn’t been loaded yet). The file is searched on sys.path. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored. Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [[filename:]lineno [function[, condition ]]] Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as break.

clear [bpnumber [bpnumber ...]] With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [bpnumber [bpnumber ...]] Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable [bpnumber [bpnumber ...]] Enables the breakpoints specified.

ignore bpnumber [count] Sets the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition] Condition is an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

s(tep) Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).
**n(ext)**  Continue execution until the next line in the current function is reached or it returns. (The difference between ‘next’ and ‘step’ is that ‘step’ stops inside a called function, while ‘next’ executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

**r(eturn)**  Continue execution until the current function returns.

**c(ont(inue))**  Continue execution, only stop when a breakpoint is encountered.

**l(ist)**  List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

**a(rgs)**  Print the argument list of the current function.

**p expression**  Evaluate the expression in the current context and print its value. (Note: ‘print’ can also be used, but is not a debugger command — this executes the Python print statement.)

**alias**  Creates an alias called name that executes command. The command must not be enclosed in quotes. Replaceable parameters can be indicated by ‘%1’, ‘%2’, and so on, while ‘%*’ is replaced by all the parameters. If no command is given, the current alias for name is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands can be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the `.pdbrc` file):

```python
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

**unalias name**  Deletes the specified alias.

**[!]statement**  Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a ‘global’ command on the same line, e.g.:

```python
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

**q(uit)**  Quit from the debugger. The program being executed is aborted.

### 9.2 How It Works

Some changes were made to the interpreter:

- `sys.settrace(func)` sets the global trace function
- there can also be a local trace function (see later)
Trace functions have three arguments: frame, event, and arg. frame is the current stack frame. event is a string: ‘call’, ‘line’, ‘return’ or ‘exception’. arg depends on the event type.

The global trace function is invoked (with event set to ‘call’) whenever a new local scope is entered; it should return a reference to the local trace function to be used that scope, or None if the scope shouldn’t be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or None to turn off tracing in that scope.

Instance methods are accepted (and very useful!) as trace functions.

The events have the following meaning:

‘call’ A function is called (or some other code block entered). The global trace function is called; arg is the argument list to the function; the return value specifies the local trace function.

‘line’ The interpreter is about to execute a new line of code (sometimes multiple line events on one line exist). The local trace function is called; arg in None; the return value specifies the new local trace function.

‘return’ A function (or other code block) is about to return. The local trace function is called; arg is the value that will be returned. The trace function’s return value is ignored.

‘exception’ An exception has occurred. The local trace function is called; arg is a triple (exception, value, traceback); the return value specifies the new local trace function.

Note that as an exception is propagated down the chain of callers, an ‘exception’ event is generated at each level.

For more information on code and frame objects, refer to the Python Reference Manual.
The Python Profiler

Copyright © 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind.¹

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The profiler was written after only programming in Python for 3 weeks. As a result, it is probably clumsy code, but I don’t know for sure yet 'cause I’m a beginner :-). I did work hard to make the code run fast, so that profiling would be a reasonable thing to do. I tried not to repeat code fragments, but I’m sure I did some stuff in really awkward ways at times. Please send suggestions for improvements to: jar@netscape.com. I won’t promise any support. ...but I’d appreciate the feedback.

10.1 Introduction to the profiler

A profiler is a program that describes the run time performance of a program, providing a variety of statistics. This documentation describes the profiler functionality provided in the modules profile and pstats. This profiler provides deterministic profiling of any Python programs. It also provides a series of report generation tools to allow users to rapidly examine the results of a profile operation.

10.2 How Is This Profiler Different From The Old Profiler?

(This section is of historical importance only; the old profiler discussed here was last seen in Python 1.1.)

The big changes from old profiling module are that you get more information, and you pay less CPU time. It’s not a trade-off, it’s a trade-up.

¹Updated and converted to LaTeX by Guido van Rossum. The references to the old profiler are left in the text, although it no longer exists.
To be specific:

**Bugs removed:** Local stack frame is no longer molested, execution time is now charged to correct functions.

**Accuracy increased:** Profiler execution time is no longer charged to user’s code, calibration for platform is supported, file reads are not done by profiler during profiling (and charged to user’s code!).

**Speed increased:** Overhead CPU cost was reduced by more than a factor of two (perhaps a factor of five), lightweight profiler module is all that must be loaded, and the report generating module (**pstats**) is not needed during profiling.

**Recursive functions support:** Cumulative times in recursive functions are correctly calculated; recursive entries are counted.

**Large growth in report generating UI:** Distinct profiles runs can be added together forming a comprehensive report; functions that import statistics take arbitrary lists of files; sorting criteria is now based on keywords (instead of 4 integer options); reports shows what functions were profiled as well as what profile file was referenced; output format has been improved.

### 10.3 Instant Users Manual

This section is provided for users that “don’t want to read the manual.” It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile an application with a main entry point of `foo()`, you would add the following to your module:

```python
import profile
profile.run('foo()')
```

The above action would cause `foo()` to be run, and a series of informative lines (the profile) to be printed. The above approach is most useful when working with the interpreter. If you would like to save the results of a profile into a file for later examination, you can supply a file name as the second argument to the `run()` function:

```python
import profile
profile.run('foo()', 'fooprof')
```

The file `profile.py` can also be invoked as a script to profile another script. For example:

```bash
python /usr/local/lib/python1.5/profile.py myscript.py
```

When you wish to review the profile, you should use the methods in the **pstats** module. Typically you would load the statistics data as follows:

```python
import pstats
p = pstats.Stats('fooprof')
```

The class **Stats** (the above code just created an instance of this class) has a variety of methods for manipulating and printing the data that was just read into `p`. When you ran `profile.run()` above, what was printed was the result of three method calls:
The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```python
p.strip_dirs().sort_stats(-1).print_stats()
```

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```python
p.sort_stats('name')
p.print_stats()
```

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```python
p.sort_stats('name')
p.print_stats()
```

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

```python
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```python
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```python
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```python
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (’cause they are spelled with ‘__init__’ in them). As one final example, you could try:

```python
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: ‘.5’) of its original size, then only lines containing init are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (’p’ is still sorted according to the last criteria) do:

```python
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you’re going to have to read the manual, or guess what the following functions do:
Invoked as a script, the `pstats` module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using `cmd`) and interactive help.

### 10.4 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user’s code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify “hot loops” that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

### 10.5 Reference Manual

The primary entry point for the profiler is the global function `profile.run()`. It is typically used to create any profile information. The reports are formatted and printed using methods of the class `pstats.Stats`. The following is a description of all of these standard entry points and functions. For a more in-depth view of some of the code, consider reading the later section on Profiler Extensions, which includes discussion of how to derive “better” profilers from the classes presented, or reading the source code for these modules.

#### `run` *(string[, filename][, ...]*)

This function takes a single argument that has can be passed to the `exec` statement, and an optional file name. In all cases this routine attempts to `exec` its first argument, and gather profiling statistics from the execution. If no file name is present, then this function automatically prints a simple profiling report, sorted by the standard name string (file/line/function-name) that is presented in each line. The following is a typical output from such a call:

```python
main()
2706 function calls (2004 primitive calls) in 4.504 CPU seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      2  0.006    0.003  0.953    0.477  pobject.py:75(save_objects)
    43/3  0.533  0.012  0.749  0.250  pobject.py:99(evaluate)
...```

---

---
The first line indicates that this profile was generated by the call:
`profile.run('main()')`, and hence the exec'ed string is `'main()'`. The second line indicates that 2706 calls were monitored. Of those calls, 2004 were primitive. We define primitive to mean that the call was not induced via recursion. The next line: Ordered by: standard name, indicates that the text string in the far right column was used to sort the output. The column headings include:

- `ncalls` for the number of calls,
- `tottime` for the total time spent in the given function (and excluding time made in calls to sub-functions),
- `percall` is the quotient of `tottime` divided by `ncalls`
- `cumtime` is the total time spent in this and all subfunctions (i.e., from invocation till exit). This figure is accurate even for recursive functions.
- `percall` is the quotient of `cumtime` divided by primitive calls
- `filename:lineno(function)` provides the respective data of each function

When there are two numbers in the first column (e.g.: `'43/3'`), then the latter is the number of primitive calls, and the former is the actual number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Analysis of the profiler data is done using this class from the `pstats` module:

```python
class Stats(filename[, ...])
```

This class constructor creates an instance of a “statistics object” from a `filename` (or set of filenames). Stats objects are manipulated by methods, in order to print useful reports.

The file selected by the above constructor must have been created by the corresponding version of `profile`. To be specific, there is no file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers (e.g., the old system profiler).

If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing Stats object, the `add()` method can be used.

### 10.5.1 The Stats Class

Stats objects have the following methods:

- **strip_dirs()**
  - This method for the Stats class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a “random” order, as it was just after object initialization and loading. If strip_dirs() causes two function names to be indistinguishable (i.e., they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

- **add(filename[, ...])**
  - This method of the Stats class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

- **sort_stats(key[, ...])**
  - This method modifies the Stats object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: ‘time’ or ‘name’).
  - When more than one key is provided, then additional keys are used as secondary criteria when the there is equality in all keys selected before them. For example, ‘sort_stats(‘name’, ‘file’)’ will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.
Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

<table>
<thead>
<tr>
<th>Valid Arg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'calls'</td>
<td>call count</td>
</tr>
<tr>
<td>'cumulative'</td>
<td>cumulative time</td>
</tr>
<tr>
<td>'file'</td>
<td>file name</td>
</tr>
<tr>
<td>'module'</td>
<td>file name</td>
</tr>
<tr>
<td>'pcalls'</td>
<td>primitive call count</td>
</tr>
<tr>
<td>'line'</td>
<td>line number</td>
</tr>
<tr>
<td>'name'</td>
<td>function name</td>
</tr>
<tr>
<td>'nfl'</td>
<td>name/file/line</td>
</tr>
<tr>
<td>'stdname'</td>
<td>standard name</td>
</tr>
<tr>
<td>'time'</td>
<td>internal time</td>
</tr>
</tbody>
</table>

Note that all sorts on statistics are in descending order (placing most time consuming items first), whereas name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between 'nfl' and 'stdname' is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, 'nfl' does a numeric compare of the line numbers. In fact, `sort_stats('nfl')` is the same as `sort_stats('name', 'file', 'line')`.

For compatibility with the old profiler, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as 'stdname', 'calls', 'time', and 'cumulative' respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()

This method for the Stats class reverses the ordering of the basic list within the object. This method is provided primarily for compatibility with the old profiler. Its utility is questionable now that ascending vs descending order is properly selected based on the sort key of choice.

print_stats(restriction[...])

This method for the Stats class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed; as of Python 1.5b1, this uses the Perl-style regular expression syntax defined by the `re` module). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename '.foo:]. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names '.foo:', and then proceed to only print the first 10% of them.

print_callers(restrictions[...])

This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the definition of the restricting argument is also identical. For convenience, a number is shown in parentheses after each caller to show how many times
this specific call was made. A second non-parenthesized number is the cumulative time spent in the function at
the right.

\texttt{print\_callees}(\texttt{restrictions}[...])

This method for the \texttt{Stats} class prints a list of all function that were called by the indicated function. Aside
from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to
the \texttt{print\_callers()} method.

\texttt{ignore()}

\textbf{Deprecated since release 1.5.1.} This is not needed in modern versions of Python.\textsuperscript{2}

10.6 \textbf{Limitations}

There are two fundamental limitations on this profiler. The first is that it relies on the Python interpreter to dispatch
\texttt{call}, \texttt{return}, and \texttt{exception} events. Compiled C code does not get interpreted, and hence is “invisible” to the profiler.
All time spent in C code (including built-in functions) will be charged to the Python function that invoked the C code.
If the C code calls out to some native Python code, then those calls will be profiled properly.

The second limitation has to do with accuracy of timing information. There is a fundamental problem with determin-
istic profilers involving accuracy. The most obvious restriction is that the underlying “clock” is only ticking at a rate
(typically) of about .001 seconds. Hence no measurements will be more accurate that that underlying clock. If enough
measurements are taken, then the “error” will tend to average out. Unfortunately, removing this first error induces a
second source of error...

The second problem is that it “takes a while” from when an event is dispatched until the profiler’s call to get the time
actually \texttt{gets} the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the
time that the clock’s value was obtained (and then squirreled away), until the user’s code is once again executing. As
a result, functions that are called many times, or call many functions, will typically accumulate this error. The error
that accumulates in this fashion is typically less than the accuracy of the clock (i.e., less than one clock tick), but it
\textbf{can} accumulate and become very significant. This profiler provides a means of calibrating itself for a given platform
so that this error can be probabilistically (i.e., on the average) removed. After the profiler is calibrated, it will be more
accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally
low, and the gods of probability work against you :-). ) \textbf{Do not} be alarmed by negative numbers in the profile. They
\textbf{should} \textit{only} appear if you have calibrated your profiler, and the results are actually better than without calibration.

10.7 \textbf{Calibration}

The profiler class has a hard coded constant that is added to each event handling time to compensate for the overhead
of calling the time function, and socking away the results. The following procedure can be used to obtain this constant
for a given platform (see discussion in section Limitations above).

\begin{verbatim}
import profile
pr = profile.Profile()
print pr.calibrate(100)
print pr.calibrate(100)
print pr.calibrate(100)
\end{verbatim}

The argument to \texttt{calibrate()} is the number of times to try to do the sample calls to get the CPU times. If your
computer is \textit{very} fast, you might have to do:

\textsuperscript{2}This was once necessary, when Python would print any unused expression result that was not \texttt{None}. The method is still defined for backward
compatibility.
pr.calibrate(1000)

or even:

pr.calibrate(10000)

The object of this exercise is to get a fairly consistent result. When you have a consistent answer, you are ready to use
that number in the source code. For a Sun Sparcstation 1000 running Solaris 2.3, the magical number is about .00053.
If you have a choice, you are better off with a smaller constant, and your results will “less often” show up as negative
in profile statistics.

The following shows how the trace_dispatch() method in the Profile class should be modified to install the calibration
constant on a Sun Sparcstation 1000:

```python
def trace_dispatch(self, frame, event, arg):
    t = self.timer()
    t = t[0] + t[1] - self.t - .00053 # Calibration constant

    if self.dispatch[event](frame,t):
        t = self.timer()
        self.t = t[0] + t[1]
    else:
        r = self.timer()
        self.t = r[0] + r[1] - t # put back unrecorded delta
    return
```

Note that if there is no calibration constant, then the line containing the calibration constant should simply say:

```python
t = t[0] + t[1] - self.t # no calibration constant
```

You can also achieve the same results using a derived class (and the profiler will actually run equally fast!!), but the
above method is the simplest to use. I could have made the profiler “self calibrating”, but it would have made the
initialization of the profiler class slower, and would have required some very fancy coding, or else the use of a variable
where the constant ‘.00053’ was placed in the code shown. This is a VERY critical performance section, and there
is no reason to use a variable lookup at this point, when a constant can be used.

10.8 Extensions — Deriving Better Profilers

The Profile class of module profile was written so that derived classes could be developed to extend the profiler.
Rather than describing all the details of such an effort, I’ll just present the following two examples of derived classes
that can be used to do profiling. If the reader is an avid Python programmer, then it should be possible to use these as
a model and create similar (and perchance better) profile classes.

If all you want to do is change how the timer is called, or which timer function is used, then the basic class has an
option for that in the constructor for the class. Consider passing the name of a function to call into the constructor:

```python
pr = profile.Profile(your_time_func)
```
The resulting profiler will call your_time_func() instead of os.times(). The function should return either a single number or a list of numbers (like what os.times() returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose. For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (os.times() is pretty bad, 'cause it returns a tuple of floating point values, so all arithmetic is floating point in the profiler!). If you want to substitute a better timer in the cleanest fashion, you should derive a class, and simply put in the replacement dispatch method that better handles your timer call, along with the appropriate calibration constant :-).

10.8.1 OldProfile Class

The following derived profiler simulates the old style profiler, providing errant results on recursive functions. The reason for the usefulness of this profiler is that it runs faster (i.e., less overhead) than the old profiler. It still creates all the caller stats, and is quite useful when there is no recursion in the user’s code. It is also a lot more accurate than the old profiler, as it does not charge all its overhead time to the user’s code.
class OldProfile(Profile):

def trace_dispatch_exception(self, frame, t):
    rt, rtt, rct, rfn, rframe, rcur = self.cur
    if rcur and not rframe is frame:
        return self.trace_dispatch_return(rframe, t)
    return 0

def trace_dispatch_call(self, frame, t):
    fn = 'frame.f_code'

    self.cur = (t, 0, 0, fn, frame, self.cur)
    if self.timings.has_key(fn):
        tt, ct, callers = self.timings[fn]
        self.timings[fn] = tt, ct, callers
    else:
        self.timings[fn] = 0, 0, {}
    return 1

def trace_dispatch_return(self, frame, t):
    rt, rtt, rct, rfn, frame, rcur = self.cur
    rtt = rtt + t
    sft = rtt + rct

    pt, ptt, pct, pfn, pframe, pcur = rcur
    self.cur = pt, ptt+rt, pct+sft, pfn, pframe, pcur

    tt, ct, callers = self.timings[rfn]
    if callers.has_key(pfn):
        callers[pfn] = callers[pfn] + 1
    else:
        callers[pfn] = 1
    self.timings[rfn] = tt+rtt, ct + sft, callers
    return 1

def snapshot_stats(self):
    self.stats = {}
    for func in self.timings.keys():
        tt, ct, callers = self.timings[func]
        nor_func = self.func_normalize(func)
        nor_callers = {}
        nc = 0
        for func_caller in callers.keys():
            nor_callers[self.func_normalize(func_caller)] = \
            callers[func_caller]
            nc = nc + callers[func_caller]
        self.stats[nor_func] = nc, nc, tt, ct, nor_callers

10.8.2 HotProfile Class

This profiler is the fastest derived profile example. It does not calculate caller-callee relationships, and does not calculate cumulative time under a function. It only calculates time spent in a function, so it runs very quickly (re: very low overhead). In truth, the basic profiler is so fast, that is probably not worth the savings to give up the data, but this
class still provides a nice example.

class HotProfile(Profile):
    def trace_dispatch_exception(self, frame, t):
        rt, rtt, rfn, rframe, rcur = self.cur
        if rcur and not rframe is frame:
            return self.trace_dispatch_return(rframe, t)
        return 0
    def trace_dispatch_call(self, frame, t):
        self.cur = (t, 0, frame, self.cur)
        return 1
    def trace_dispatch_return(self, frame, t):
        rt, rtt, frame, rcur = self.cur
        rfn = 'frame.f_code'
        pt, ptt, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pframe, pcur
        if self.timings.has_key(rfn):
            nc, tt = self.timings[rfn]
            self.timings[rfn] = nc + 1, rt + rtt + tt
        else:
            self.timings[rfn] = 1, rt + rtt
        return 1
    def snapshot_stats(self):
        self.stats = {}
        for func in self.timings.keys():
            nc, tt = self.timings[func]
            nor_func = self.func_normalize(func)
            self.stats[nor_func] = nc, nc, tt, 0, {}
The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module `socket`, which is currently supported on most popular platforms. Here is an overview:

- **webbrowser** — Easy-to-use controller for Web browsers.
- **cgi** — Common Gateway Interface support, used to interpret forms in server-side scripts.
- **urllib** — Open an arbitrary network resource by URL (requires sockets).
- **urllib2** — An extensible library for opening URLs using a variety of protocols.
- **httplib** — HTTP protocol client (requires sockets).
- **ftplib** — FTP protocol client (requires sockets).
- **gopherlib** — Gopher protocol client (requires sockets).
- **poplib** — POP3 protocol client (requires sockets).
- **imaplib** — IMAP4 protocol client (requires sockets).
- **nntplib** — NNTP protocol client (requires sockets).
- **smtplib** — SMTP protocol client (requires sockets).
- **telnetlib** — Telnet client class.
- **urlparse** — Parse URLs into components.
- **SocketServer** — A framework for network servers.
- **BaseHTTPServer** — Basic HTTP server (base class for SimpleHTTPServer and CGIHTTPServer).
- **SimpleHTTPServer** — This module provides a basic request handler for HTTP servers.
- **CGIHTTPServer** — This module provides a request handler for HTTP servers which can run CGI scripts.
- **Cookie** — Support for HTTP state management (cookies).
- **asyncore** — A base class for developing asynchronous socket handling services.

### 11.1 webbrowser — Convenient Web-browser controller

The `webbrowser` module provides a very high-level interface to allow displaying Web-based documents to users. The controller objects are easy to use and are platform-independent. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under UNIX, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn’t available. If text-mode browsers are used, the calling process will block until the user exits the browser.

Under UNIX, if the environment variable BROWSER exists, it is interpreted to override the platform default list of browsers, as a colon-separated list of browsers to try in order. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for the `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch.

For non-UNIX platforms, or when X11 browsers are available on UNIX, the controlling process will not wait for the
user to finish with the browser, but allow the browser to maintain its own window on the display.

The following exception is defined:

**exception Error**

Exception raised when a browser control error occurs.

The following functions are defined:

**open (url[, new=0][, autoraise=1])**

Display url using the default browser. If new is true, a new browser window is opened if possible. If autoraise is true, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

**open_new (url)**

Open url in a new window of the default browser, if possible, otherwise, open url in the only browser window. (This entry point is deprecated and may be removed in 2.1.)

**get ([name])**

Return a controller object for the browser type name. If name is empty, return a controller for a default browser appropriate to the caller's environment.

**register (name, constructor[, instance])**

Register the browser type name. Once a browser type is registered, the get() function can return a controller for that browser type. If instance is not provided, or is None, constructor will be called without parameters to create an instance when needed. If instance is provided, constructor will never be called, and may be None.

This entry point is only useful if you plan to either set the BROWSER variable or call get with a nonempty argument matching the name of a handler you declare.

A number of browser types are predefined. This table gives the type names that may be passed to the get() function and the corresponding instantiations for the controller classes, all defined in this module.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Class Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'mozilla'</td>
<td>Netscape('mozilla')</td>
<td></td>
</tr>
<tr>
<td>'netscape'</td>
<td>Netscape('netscape')</td>
<td></td>
</tr>
<tr>
<td>'mosaic'</td>
<td>GenericBrowser('mosaic %s &amp;')</td>
<td></td>
</tr>
<tr>
<td>'kfm'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'grail'</td>
<td>Grail()</td>
<td></td>
</tr>
<tr>
<td>'links'</td>
<td>GenericBrowser('links %s')</td>
<td></td>
</tr>
<tr>
<td>'lynx'</td>
<td>GenericBrowser('lynx %s')</td>
<td></td>
</tr>
<tr>
<td>'w3m'</td>
<td>GenericBrowser('w3m %s')</td>
<td></td>
</tr>
<tr>
<td>'windows-default'</td>
<td>WindowsDefault</td>
<td>(2)</td>
</tr>
<tr>
<td>'internet-config'</td>
<td>InternetConfig</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Notes:

(1) “Konqueror” is the file manager for the KDE desktop environment for UNIX, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the KDEDIR variable is not sufficient. Note also that the name “kfm” is used even when using the konqueror command with KDE 2 — the implementation selects the best strategy for running Konqueror.

(2) Only on Windows platforms; requires the common extension modules win32api and win32con.

(3) Only on MacOS platforms; requires the standard MacPython ic module, described in the *Macintosh Library Modules* manual.
11.1.1 Browser Controller Objects

Browser controllers provide two methods which parallel two of the module-level convenience functions:

- **open**(url[, new])
  Display url using the browser handled by this controller. If new is true, a new browser window is opened if possible.

- **open_new**(url)
  Open url in a new window of the browser handled by this controller, if possible, otherwise, open url in the only browser window. (This method is deprecated and may be removed in 2.1.)

11.2 **cgi — Common Gateway Interface support.**

Support module for CGI (Common Gateway Interface) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

11.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server’s special ‘cgi-bin’ directory. The HTTP server places all sorts of information about the request (such as the client’s hostname, the requested URL, the query string, and lots of other goodies) in the script’s shell environment, executes the script, and sends the script’s output back to the client.

The script’s input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the “query string” part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it — Grail 0.3 and Netscape 2.0 do).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```python
print "Content-Type: text/html"  # HTML is following
print ""  # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here’s Python code that prints a simple piece of HTML:

```python
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```

11.2.2 Using the cgi module

Begin by writing ‘import cgi’. Do not use ‘from cgi import *’ — the module defines all sorts of names for its own use or for backward compatibility that you don’t want in your namespace.

11.2. **cgi — Common Gateway Interface support.**
It's best to use the `FieldStorage` class. The other classes defined in this module are provided mostly for backward compatibility. Instantiate it exactly once, without arguments. This reads the form contents from standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary, and also supports the standard dictionary methods `has_key()` and `keys()`. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide the optional `keep_blank_values` argument when creating the `FieldStorage` instance.

For instance, the following code (which assumes that the `Content-Type` header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```python
form = cgi.FieldStorage()
form_ok = 0
if form.has_key("name") and form.has_key("addr"):
    form_ok = 1
if not form_ok:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name: ", form["name"].value
print "<p>addr: ", form["addr"].value
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (i.e., when your HTML form contains multiple fields with the same name), use the `type()` function to determine whether you have a single instance or a list of instances. For example, here's code that concatenates any number of username fields, separated by commas:

```python
value = form.getvalue("username", ")
if type(value) is type([]):
    # Multiple username fields specified
    usernames = ",".join(value)
else:
    # Single or no username field specified
    usernames = value
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as a string. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data at leisure from the `file` attribute:
if fileitem.file:
    # It’s an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive multipart/* encoding). When this occurs, the item will be a dictionary-like FieldStorage item. This can be determined by testing its type attribute, which should be multipart/form-data (or perhaps another MIME type matching multipart/*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the “old” format (as the query string or as a single data part of type application/x-www-form-urlencoded), the items will actually be instances of the class MiniFieldStorage. In this case, the list, file, and filename attributes are always None.

11.2.3 Old classes

These classes, present in earlier versions of the cgi module, are still supported for backward compatibility. New applications should use the FieldStorage class.

SvFormContentDict stores single value form content as dictionary; it assumes each field name occurs in the form only once.

FormContentDict stores multiple value form content as a dictionary (the form items are lists of values). Useful if your form contains multiple fields with the same name.

Other classes (FormContent, InterpFormContentDict) are present for backwards compatibility with really old applications only. If you still use these and would be inconvenienced when they disappeared from a next version of this module, drop me a note.

11.2.4 Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

```
parse(fp)

Parse a query in the environment or from a file (default sys.stdin).
```

```
parse_qs(qs[, keep_blank_values, strict_parsing])

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument keep_blank_values is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument strict_parsing is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.
```

```
parse_qsl(qs[, keep_blank_values, strict_parsing])

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a list of name, value pairs.
```

11.2. cgi — Common Gateway Interface support. 279
The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.

**parse_multipart** *(fp, pdict)*  
Parse input of type multipart/form-data (for file uploads). Arguments are *fp* for the input file and *pdict* for a dictionary containing other parameters in the Content-Type header.

Returns a dictionary just like parse_qs() keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the FieldStorage class instead which is much more flexible.

Note that this does not parse nested multipart parts — use FieldStorage for that.

**parse_header** *(string)*  
Parse a MIME header (such as Content-Type) into a main value and a dictionary of parameters.

**test** ()  
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

**print_environ** ()  
Format the shell environment in HTML.

**print_form** *(form)*  
Format a form in HTML.

**print_directory** ()  
Format the current directory in HTML.

**print_environ_usage** ()  
Print a list of useful (used by CGI) environment variables in HTML.

**escape** *(s[, quote ])*  
Convert the characters ‘&’, ‘<’ and ‘>’ in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the double quote character (""") is also translated; this helps for inclusion in an HTML attribute value, e.g. in \<A HREF="...">.

### 11.2.5 Caring about security

There’s one important rule: if you invoke an external program (e.g. via the os.system() or os.popen() functions), make very sure you don’t pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn’t have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

### 11.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory ‘cgi-bin’ in the server tree.

Make sure that your script is readable and executable by “others”; the UNIX file mode should be 0755 octal (use ‘chmod 0755 filename’). Make sure that the first line of the script contains #! starting in column 1 followed by the pathname of the Python interpreter, for instance:
Make sure the Python interpreter exists and is executable by “others”.

Make sure that any files your script needs to read or write are readable or writable, respectively, by “others” — their mode should be 0644 for readable and 0666 for writable. This is because, for security reasons, the HTTP server executes your script as user “nobody”, without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server’s cgi-bin directory) and the set of environment variables is also different from what you get at login. In particular, don’t count on the shell’s search path for executables (PATH) or the Python module search path (PYTHONPATH) to be set to anything interesting.

If you need to load modules from a directory which is not on Python’s default module search path, you can change the path in your script, before importing other modules, e.g.:

```python
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-UNIX systems will vary; check your HTTP server’s documentation (it will usually have a section on CGI scripts).

11.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There’s one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won’t execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

11.2.8 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it’s installed in the standard ‘cgi-bin’ directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error (e.g. 500), there’s an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as “addr” with value “At Home” and “name” with value “Joe Blow”), the ‘cgi.py’ script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the `cgi` module’s `test()` function from your script: replace its main code with the single statement

```
11.2. cgi — Common Gateway Interface support. 281
```
cgi.test()

This should produce the same results as those gotten from installing the ‘cgi.py’ file itself.

When an ordinary Python script raises an unhandled exception (e.g. because of a typo in a module name, a file that can’t be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server’s log file, or be discarded altogether.

Fortunately, once you have managed to get your script to execute some code, it is easy to catch exceptions and cause a traceback to be printed. The test() function below in this module is an example. Here are the rules:

1. Import the traceback module before entering the try ... except statement
2. Assign sys.stderr to be sys.stdout
3. Make sure you finish printing the headers and the blank line early
4. Wrap all remaining code in a try ... except statement
5. In the except clause, call traceback.print_exc()

For example:

```python
import sys
import traceback
print "Content-Type: text/html"
print
sys.stderr = sys.stdout
try:
    ...your code here...
except:
    print "\n\n<PRE>"
    traceback.print_exc()
```

Notes: The assignment to sys.stderr is needed because the traceback prints to sys.stderr. The print "\n\n<PRE>" statement is necessary to disable the word wrapping in HTML.

If you suspect that there may be a problem in importing the traceback module, you can use an even more robust approach (which only uses built-in modules):

```python
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will readable.
11.2.9 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client’s display while the script is running.
- Check the installation instructions above.
- Check the HTTP server’s log files. (‘tail -f logfile’ in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like ‘python script.py’.
- When using any of the debugging techniques, don’t forget to add ‘import sys’ to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — PATH is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by every user on the system.
- Don’t try to give a CGI script a set-uid mode. This doesn’t work on most systems, and is a security liability as well.

11.3 urllib — Open arbitrary resources by URL

This module provides a high-level interface for fetching data across the World-Wide Web. In particular, the urlopen function is similar to the built-in function open, but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply — it can only open URLs for reading, and no seek operations are available.

It defines the following public functions:

urlopen(url[, data])

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has ‘file:’ as its scheme identifier, this opens a local file; otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made, or if the server returns an error code, the IOError exception is raised. If all went well, a file-like object is returned. This supports the following methods: read(), readline(), readlines(), fileno(), close(), info() and geturl().

Except for the info() and geturl() methods, these methods have the same interface as for file objects — see section 2.1.7 in this manual. (It is not a built-in file object, however, so it can’t be used at those few places where a true built-in file object is required.)

The info() method returns an instance of the class mimetools.Message containing meta-information associated with the URL. When the method is HTTP, these headers are those returned by the server at the head of the retrieved HTML page (including Content-Length and Content-Type). When the method is FTP, a Content-Length header will be present if (as is now usual) the server passed back a file length in response to the FTP retrieval request. When the method is local-file, returned headers will include a Date representing the file’s last-modified time, a Content-Length giving file size, and a Content-Type containing a guess at the file’s type. See also the description of the mimetools module.

The geturl() method returns the real URL of the page. In some cases, the HTTP server redirects a client to another URL. The urlopen() function handles this transparently, but in some cases the caller needs to know which URL the client was redirected to. The geturl() method can be used to get at this redirected URL.

If the url uses the ‘http:’ scheme identifier, the optional data argument may be given to specify a POST request (normally the request type is GET). The data argument must in standard application/x-www-form-urlencoded format; see the urlopen() function below.

11.3. urllib — Open arbitrary resources by URL 283
The `urlopen()` function works transparently with proxies which do not require authentication. In a UNIX or Windows environment, set the `http_proxy`, `ftp_proxy` or `gopher_proxy` environment variables to a URL that identifies the proxy server before starting the Python interpreter. For example (the `%` is the command prompt):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
```

In a Macintosh environment, `urlopen()` will retrieve proxy information from Internet Config.

Proxies which require authentication for use are not currently supported; this is considered an implementation limitation.

```
urlretrieve(url[, filename[, reporthook[, data]]])
```

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple `(filename, headers)` where `filename` is the local file name under which the object can be found, and `headers` is either `None` (for a local object) or whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object, possibly cached). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a temporary file with a generated name). The third argument, if present, is a hook function that will be called once on establishment of the network connection and once after each block read thereafter. The hook will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

If the `url` uses the ‘http:’ scheme identifier, the optional `data` argument may be given to specify a POST request (normally the request type is GET). The `data` argument must in standard `application/x-www-form-urlencoded` format; see the `urlencode()` function below.

```
urlcleanup()
```

Clear the cache that may have been built up by previous calls to `urlretrieve()`.

```
quote(string[, safe])
```

Replace special characters in `string` using the ‘%xx’ escape. Letters, digits, and the characters ‘-_,.’ are never quoted. The optional `safe` parameter specifies additional characters that should not be quoted — its default value is ‘ ‘.

Example: `quote(’/~connolly/’)` yields ‘/%7econnolly/’.

```
quote_plus(string[, safe])
```

Like `quote()`, but also replaces spaces by plus signs, as required for quoting HTML form values. Plus signs in the original string are escaped unless they are included in `safe`.

```
unquote(string)
```

Replace ‘%xx’ escapes by their single-character equivalent.

Example: `unquote(’/%7Econnolly/’)` yields ‘/~connolly/’.

```
unquote_plus(string)
```

Like `unquote()`, but also replaces plus signs by spaces, as required for unquoting HTML form values.

```
urlencode(query[, doseq])
```

Convert a mapping object or a sequence of two-element tuples to a “url-encoded” string, suitable to pass to `urlopen()` above as the optional `data` argument. This is useful to pass a dictionary of form fields to a POST request. The resulting string is a series of `key=value` pairs separated by ‘&’ characters, where both `key` and `value` are quoted using `quote_plus()` above. If the optional parameter `doseq` is present and evaluates to true, individual `key=value` pairs are generated for each element of the sequence. When a sequence of two-element tuples is used as the `query` argument, the first element of each tuple is a key and the second is a value. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.
The public functions `urlopen()` and `urlretrieve()` create an instance of the `FancyURLopener` class and use it to perform their requested actions. To override this functionality, programmers can create a subclass of `URLopener` or `FancyURLopener`, then assign that an instance of that class to the `urllib._urlopener` variable before calling the desired function. For example, applications may want to specify a different user-agent header than `URLopener` defines. This can be accomplished with the following code:

```python
class AppURLopener(urllib.FancyURLopener):
    def __init__(self, *args):
        self.version = "App/1.7"
        apply(urllib.FancyURLopener.__init__, (self,) + args)

urllib._urlopener = AppURLopener()
```

### class `URLopener` ([`proxies`, `**x509`])
Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than ‘http’, ‘ftp’, ‘gopher’ or ‘file’, you probably want to use `FancyURLopener`.

By default, the `URLopener` class sends a user-agent header of ‘urllib/VVV’, where VVV is the `urllib` version number. Applications can define their own user-agent header by subclassing `URLopener` or `FancyURLopener` and setting the instance attribute `version` to an appropriate string value before the `open()` method is called.

Additional keyword parameters, collected in `x509`, are used for authentication with the ‘https’ scheme. The keywords `key_file` and `cert_file` are supported; both are needed to actually retrieve a resource at an ‘https’ URL.

### class `FancyURLopener` (...)
`FancyURLopener` subclasses `URLopener` providing default handling for the following HTTP response codes: 301, 302 or 401. For 301 and 302 response codes, the `location` header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For 301 and 302 response codes, recursion is bounded by the value of the `maxtries` attribute, which defaults 10.

The parameters to the constructor are the same as those for `URLopener`.

**Note:** When performing basic authentication, a `FancyURLopener` instance calls its `prompt_user_passwd()` method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

**Restrictions:**

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), Gopher (but not Gopher-+), FTP, and local files.
- The caching feature of `urlretrieve()` has been disabled until I find the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can’t be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.
- The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (e.g. an image), plain text or (for example) HTML. The HTTP protocol provides type information in the `Content-Type` header.
the reply header, which can be inspected by looking at the content-type header. For the Gopher protocol, type information is encoded in the URL; there is currently no easy way to extract it. If the returned data is HTML, you can use the module htmllib to parse it.

- This module does not support the use of proxies which require authentication. This may be implemented in the future.
- Although the urllib module contains (undocumented) routines to parse and unpars URL strings, the recommended interface for URL manipulation is in module urlparse.

11.3.1 URLopener Objects

URLopener and FancyURLopener objects have the following attributes.

**open** *(fullurl[, data]*)

Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of urlopen().

**open_unknown** *(fullurl[, data]*)

Overridable interface to open unknown URL types.

**retrieve** *(url[, filename[, reporthook[, data]]]*)

Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either a mimetools.Message object containing the response headers (for remote URLs) or None (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of tempfile.mktemp() with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters. It will be called after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the ‘http’ scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must in standard application/x-www-form-urlencoded format; see the urlencode() function below.

**version**

Variable that specifies the user agent of the opener object. To get urllib to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

The FancyURLopener class offers one additional method that should be overloaded to provide the appropriate behavior:

**prompt_user_passwd** *(host, realm)*

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, *(user, password)*, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

11.3.2 Examples

Here is an example session that uses the ‘GET’ method to retrieve a URL containing parameters:
>>> import urllib
>>> params = urllib.urlencode({’spam’: 1, ’eggs’: 2, ’bacon’: 0})
>>> f = urllib.urlopen(’http://www.musical.com/cgi-bin/query?%s’ % params)
>>> print f.read()

The following example uses the ’POST’ method instead:

>>> import urllib
>>> params = urllib.urlencode({’spam’: 1, ’eggs’: 2, ’bacon’: 0})
>>> f = urllib.urlopen(’http://www.musical.com/cgi-bin/query’, params)
>>> print f.read()

11.4 urllib2 — extensible library for opening URLs

The urllib2 module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections and more.

The urllib2 module defines the following functions:

urlopen(\url[, \data])
Open the url \url, which can either a string or a Request object (currently the code checks that it really is a Request instance, or an instance of a subclass of Request).
\data should be a string, which specifies additional data to send to the server. In HTTP requests, which are the only ones that support \data, it should be a buffer in the format of application/x-www-form-urlencoded, for example one returned from urllib.urlencode().

This function returns a file-like object with two additional methods:

- \geturl() — return the URL of the resource retrieved
- \info() — return the meta-information of the page, as a dictionary-like object

Raises URLError on errors.

install_opener(opener)
Install a OpenerDirector instance as the default opener. The code does not check for a real OpenerDirector, and any class with the appropriate interface will work.

build_opener([\handler,...])
Return an OpenerDirector instance, which chains the handlers in the order given. \handlers can be either instances of BaseHandler, or subclasses of BaseHandler (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in the front of the \handlers, unless the \handlers contain them, instances of them or subclasses of them:
ProxyHandler, UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler

If the Python installation has SSL support (socket.ssl() exists), HTTPSHandler will also be added.

The following exceptions are raised as appropriate:

exception URLError
The error handlers raise when they run into a problem. It is a subclass of IOError.
exception HTTPError

A subclass of URLError, it can also function as a non-exceptional file-like return value (the same thing that urlopen() returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

exception GopherError

A subclass of URLError, this is the error raised by the Gopher handler.

The following classes are provided:

class Request (url[, data[, headers]])

This class is an abstraction of a URL request.

url should be a string which is a valid URL. For description of data see the add_data() description. headers should be a dictionary, and will be treated as if add_header() was called with each key and value as arguments.

class OpenerDirector()

The OpenerDirector class opens URLs via BaseHandlers chained together. It manages the chaining of handlers, and recovery from errors.

class BaseHandler()

This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class HTTPDefaultErrorHandler()

A class which defines a default handler for HTTP error responses; all responses are turned into HTTPError exceptions.

class HTTPRedirectHandler()

A class to handle redirections.

class ProxyHandler (proxies)

Cause requests to go through a proxy. If proxies is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables protocol_proxy.

class HTTPPasswordMgr()

Keep a database of (realm, uri) -> (user, password) mappings.

class HTTPPasswordMgrWithDefaultRealm()

Keep a database of (realm, uri) -> (user, password) mappings. A realm of None is considered a catch-all realm, which is searched if no other realm fits.

class AbstractBasicAuthHandler (password_mgr)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy.

password_mgr should be something that is compatible with HTTPPasswordMgr — supplies the documented interface above.

class HTTPBasicAuthHandler (password_mgr)

Handle authentication with the remote host. Valid password_mgr, if given, are the same as for AbstractBasicAuthHandler.

class ProxyBasicAuthHandler (password_mgr)

Handle authentication with the proxy. Valid password_mgr, if given, are the same as for AbstractBasicAuthHandler.

class AbstractDigestAuthHandler (password_mgr)

This is a mixin class, that helps with HTTP authentication, both to the remote host and to a proxy.

password_mgr should be something that is compatible with HTTPPasswordMgr — supplies the documented interface above.

class HTTPDigestAuthHandler (password_mgr)

Handle authentication with the remote host. Valid password_mgr, if given, are the same as for AbstractBasicAuthHandler.
class `ProxyDigestAuthHandler` ([`password_mngr`])
Handle authentication with the proxy. `password_mngr` should be the same as for the constructor of `AbstractDigestAuthHandler`.

class `HTTPHandler` ()
A class to handle opening of HTTP URLs.

class `HTTPSHandler` ()
A class to handle opening of HTTPS URLs.

class `FileHandler` ()
Open local files.

class `FTPHandler` ()
Open FTP URLs.

class `CacheFTPHandler` ()
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class `GopherHandler` ()
Open gopher URLs.

class `UnknownHandler` ()
A catch-all class to handle unknown URLs.

11.4.1 Request Objects

The following methods describe all of `Request`'s public interface, and so all must be overridden in subclasses.

`add_data` (`data`)
Set the `Request` data to `data` is ignored by all handlers except HTTP handlers — and there it should be an `application/x-www-form-encoded` buffer, and will change the request to be `POST` rather than `GET`.

`has_data` (`data`)
Return whether the instance has a non-`None` data.

`get_data` (`data`)
Return the instance's data.

`add_header` (`key`, `val`)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the `key` collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

`get_full_url` ()
Return the URL given in the constructor.

`get_type` ()
Return the type of the URL — also known as the scheme.

`get_host` ()
Return the host to which connection will be made.

`get_selector` ()
Return the selector — the part of the URL that is sent to the server.

`set_proxy` (`host`, `type`)
Make the request by connecting to a proxy server. The `host` and `type` will replace those of the instance, and the instance's selector will be the original URL given in the constructor.
11.4.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

**add_handler** *(handler)*

Handler should be an instance of BaseHandler. The following methods are searched, and added to the possible chains.

- `protocol_open()` — signal that the handler knows how to open protocol URLs.
- `protocol_error_type()` — signal that the handler knows how to handle type errors from protocol.

**close()**

Explicitly break cycles, and delete all the handlers. Because the OpenerDirector needs to know the registered handlers, and a handler needs to know who the OpenerDirector who called it is, there is a reference cycle. Even though recent versions of Python have cycle-collection, it is sometimes preferable to explicitly break the cycles.

**open** *(url[, data]*)

Open the given url. (which can be a request object or a string), optionally passing the given data. Arguments, return values and exceptions raised are the same as those of urlopen() (which simply calls the open() method on the default installed OpenerDirector.

**error** *(proto[, arg[....]])*

Handle an error in a given protocol. The HTTP protocol is special cased to use the code as the error. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). Return values and exceptions raised are the same as those of urlopen().

11.4.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

**add_parent** *(director)*

Add a director as parent.

**close()**

Remove any parents.

The following members and methods should be used only by classes derived from BaseHandler:

**parent**

A valid OpenerDirector, which can be used to open using a different protocol, or handle errors.

**default_open** *(req)*

This method is not defined in BaseHandler, but subclasses should define it if they want to catch all URLs.

This method, if exists, will be called by the parent OpenerDirector. It should return a file-like object as described in the return value of the open() of OpenerDirector or None. It should raise URLError, unless a truly exceptional thing happens (for example, MemoryError should not be mapped to URLError.

This method will be called before any protocol-specific open method.

**protocol_open** *(req)*

This method is not defined in BaseHandler, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent OpenerDirector. Return values should be the same as for default_open().
unknown_open(req)
This method is *not* defined in BaseHandler, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if exists, will be called by the parent OpenerDirector. Return values should be the same as for default_open().

http_error_default(req, fp, code, msg, hdrs)
This method is *not* defined in BaseHandler, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the OpenerDirector getting the error, and should not normally be called in other circumstances.

`req` will be a Request object, `fp` will be a file-like object with the HTTP error body, `code` will be the three-digit code of the error, `msg` will be the user-visible explanation of the code and `hdrs` will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of urlopen().

http_error_nnn(req, fp, code, msg, hdrs)
`nnn` should be a three-digit HTTP error code. This method is also not defined in BaseHandler, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code `nnn` occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for http_error_default().

### 11.4.4 HTTPRedirectHandler Objects

**Note:** 303 redirection is not supported by this version of urllib2.

http_error_301(req, fp, code, msg, hdrs)
Redirect to the Location: URL. This method is called by the parent OpenerDirector when getting an HTTP permanent-redirect response.

http_error_302(req, fp, code, msg, hdrs)
The same as http_error_301(), but called for the temporary-redirect response.

### 11.4.5 ProxyHandler Objects

`protocol_open(request)`
The ProxyHandler will have a method `protocol_open()` for every protocol which has a proxy in the `proxies` dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

### 11.4.6 HTTPPasswordMgr Objects

These methods are available on HTTPPasswordMgr and HTTPPasswordMgrWithDefaultRealm objects.

`add_password(realm, uri, user, passwd)`
`uri` can be either a single URI, or a sequence of URIs. `realm`, `user` and `passwd` must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for `realm` and a super-URI of any of the given URIs is given.

`find_user_password(realm, authuri)`
Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For HTTPPasswordMgrWithDefaultRealm objects, the realm `None` will be searched if the given `realm` has no matching user/password.
11.4.7  AbstractBasicAuthHandler Objects

`handle_authentication_request (authreq, host, req, headers)`
Handle an authentication request by getting user/password pair, and retrying. `authreq` should be the name of the header where the information about the realm, `host` is the host to authenticate too, `req` should be the (failed) Request object, and `headers` should be the error headers.

11.4.8  HTTPBasicAuthHandler Objects

`http_error_401 (req, fp, code, msg, hdrs)`
Retry the request with authentication info, if available.

11.4.9  ProxyBasicAuthHandler Objects

`http_error_407 (req, fp, code, msg, hdrs)`
Retry the request with authentication info, if available.

11.4.10  AbstractDigestAuthHandler Objects

`handle_authentication_request (authreq, host, req, headers)`
`authreq` should be the name of the header where the information about the realm, `host` should be the host to authenticate too, `req` should be the (failed) Request object, and `headers` should be the error headers.

11.4.11  HTTPDigestAuthHandler Objects

`http_error_401 (req, fp, code, msg, hdrs)`
Retry the request with authentication info, if available.

11.4.12  ProxyDigestAuthHandler Objects

`http_error_407 (req, fp, code, msg, hdrs)`
Retry the request with authentication information, if available.

11.4.13  HTTPHandler Objects

`http_open (req)`
Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

11.4.14  HTTPSHandler Objects

`https_open (req)`
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

11.4.15  FileHandler Objects

`file_open (req)`
Open the file locally, if there is no host name, or the host name is `localhost`. Change the protocol to `ftp` otherwise, and retry opening it using `parent`.
11.4.16 FTPHandler Objects

\texttt{ftp\_open}(\textit{req})

Open the FTP file indicated by \textit{req}. The login is always done with empty username and password.

11.4.17 CacheFTPHandler Objects

CacheFTPHandler objects are FTPHandler objects with the following additional methods:

\texttt{setTimeout}(\textit{t})

Set timeout of connections to \textit{t} seconds.

\texttt{setMaxConns}(\textit{m})

Set maximum number of cached connections to \textit{m}.

11.4.18 GopherHandler Objects

\texttt{gopher\_open}(\textit{req})

Open the gopher resource indicated by \textit{req}.

11.4.19 UnknownHandler Objects

\texttt{unknown\_open}(\textit{R})

Raise a \texttt{URLError} exception.

11.5 \texttt{httplib} — HTTP protocol client

This module defines a class which implements the client side of the HTTP protocol. It is normally not used directly — the module \texttt{urllib} uses it to handle URLs that use HTTP.

The module defines one class, \texttt{HTTP}:

\texttt{class HTTP(\{\textit{host}, port \})}

An HTTP instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form \textit{host:port}, else the default HTTP port (80) is used. If no host is passed, no connection is made, and the \texttt{connect()} method should be used to connect to a server. For example, the following calls all create instances that connect to the server at the same host and port:

\begin{verbatim}
>>> h1 = httplib.HTTP(‘www.cwi.nl’)
>>> h2 = httplib.HTTP(‘www.cwi.nl:80’)
>>> h3 = httplib.HTTP(‘www.cwi.nl’, 80)
\end{verbatim}

Once an HTTP instance has been connected to an HTTP server, it should be used as follows:

1. Make exactly one call to the \texttt{putrequest()} method.
2. Make zero or more calls to the \texttt{putheader()} method.
3. Call the \texttt{endheaders()} method (this can be omitted if step 4 makes no calls).
4. Optional calls to the \texttt{send()} method.
5. Call the \texttt{getreply()} method.
6. Call the \texttt{getfile()} method and read the data off the file object that it returns.
11.5.1 HTTP Objects

HTTP instances have the following methods:

- **set_debuglevel** *(level)*
  
  Set the debugging level (the amount of debugging output printed). The default debug level is 0, meaning no debugging output is printed.

- **connect** *(host[, port]*)
  
  Connect to the server given by *host* and *port*. See the intro for the default port. This should be called directly only if the instance was instantiated without passing a host.

- **send** *(data)*
  
  Send data to the server. This should be used directly only after the endheaders() method has been called and before getreply() has been called.

- **putrequest** *(request, selector)*
  
  This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *request* string, the *selector* string, and the HTTP version (HTTP/1.0).

- **putheader** *(header, argument[, ...]*)
  
  Send an RFC 822 style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

- **endheaders** *
  
  Send a blank line to the server, signalling the end of the headers.

- **getreply** *
  
  Complete the request by shutting down the sending end of the socket, read the reply from the server, and return a triple *(replycode, message, headers)*. Here, *replycode* is the integer reply code from the request (e.g., 200 if the request was handled properly); *message* is the message string corresponding to the reply code; and *headers* is an instance of the class mimetools.Message containing the headers received from the server. See the description of the mimetools module.

- **getfile** *
  
  Return a file object from which the data returned by the server can be read, using the read(), readline() or readlines() methods.

11.5.2 Examples

Here is an example session that uses the ‘GET’ method:

```python
>>> import httplib
>>> h = httplib.HTTP('www.cwi.nl')
>>> h.putrequest('GET', '/index.html')
>>> h.putheader('Accept', 'text/html')
>>> h.putheader('Accept', 'text/plain')
>>> h.endheaders()
>>> errcode, errmsg, headers = h.getreply()
>>> print errcode # Should be 200
>>> f = h.getfile()
>>> data = f.read() # Get the raw HTML
>>> f.close()
```

Here is an example session that shows how to ‘POST’ requests:
>>> import httpplib, urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> h = httpplib.HTTP("www.musi-cal.com:80")
>>> h.putrequest("POST", "/cgi-bin/query")
>>> h.putheader("Content-type", "application/x-www-form-urlencoded")
>>> h.putheader("Content-length", "%d" % len(params))
>>> h.putheader('Accept', 'text/plain')
>>> h.putheader('Host', 'www.musi-cal.com')
>>> h.endheaders()
>>> h.send(params)
>>> reply, msg, hdrs = h.getreply()
>>> print reply # should be 200
>>> data = h.getfile().read() # get the raw HTML

11.6 ftplib — FTP protocol client

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other ftp servers. It is also used by the module urllib to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here’s a sample session using the ftplib module:

>>> from ftplib import FTP
>>> ftp = FTP(’ftp.cwi.nl’) # connect to host, default port
>>> ftp.login() # user anonymous, passwd user@hostname
>>> ftp.retrlines(’LIST’), # list directory contents
total 24418
drwxrwsr-x 5 ftp-usr pdmaint 1536 Mar 20 09:48 .
dr-xr-srwt 105 ftp-usr pdmaint 1536 Mar 21 14:32 ..
-rw-r--r-- 1 ftp-usr pdmaint 5305 Mar 20 09:48 INDEX.

>>> ftp.retrbinary(’RETR README’, open(’README’, ’wb’)).write)
’226 Transfer complete.’
>>> ftp.quit()

The module defines the following items:

**class FTP ([host[, user[, passwd[, acct]]]])**

Return a new instance of the FTP class. When host is given, the method call connect(host) is made. When user is given, additionally the method call login(user, passwd, acct) is made (where passwd and acct default to the empty string when not given).

**all_errors**

The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed below as well as socket.error and IOError.

**exception error_reply**

Exception raised when an unexpected reply is received from the server.
**exception** `error_temp`
Exception raised when an error code in the range 400–499 is received.

**exception** `error_perm`
Exception raised when an error code in the range 500–599 is received.

**exception** `error_proto`
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

**See Also:**

Module `netrc` (section 12.17):
Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

The file ‘Tools/scripts/ftpmirror.py’ in the Python source distribution is a script that can mirror FTP sites, or portions thereof, using the `ftplib` module. It can be used as an extended example that applies this module.

### 11.6.1 FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by ‘lines’ for the text version or ‘binary’ for the binary version.

FTP instances have the following methods:

- `set_debuglevel(level)`
  Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

- `connect(host[, port])`
  Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

- `getwelcome()`
  Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

- `login([user[, passwd[, acct]])`
  Log in as the given `user`. The `passwd` and `acct` parameters are optional and default to the empty string. If no `user` is specified, it defaults to ‘anonymous’. If `user` is ‘anonymous’, the default `passwd` is `realuser@host` where `realuser` is the real user name (glanced from the LOGNAME or USER environment variable) and `host` is the hostname as returned by `socket.gethostname()`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in.

- `abort()`
  Abort a file transfer that is in progress. Using this does not always work, but it’s worth a try.

- `sendcmd(command)`
  Send a simple command string to the server and return the response string.

- `voidcmd(command)`
  Send a simple command string to the server and handle the response. Return nothing if a response code in the range 200–299 is received. Raise an exception otherwise.

- `retrbinary(command, callback[, maxblocksize[, rest]])`
Retrieve a file in binary transfer mode. `command` should be an appropriate ‘RETR’ command, i.e. ‘RETR filename’. The `callback` function is called for each block of data received, with a single string argument giving the data block. The optional `maxblocksize` argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to `callback`). A reasonable default is chosen. `rest` means the same thing as in the `transfercmd()` method.

```
retrlines (command[, callback])
```

Retrieve a file or directory listing in ASCII transfer mode. `command` should be an appropriate ‘RETR’ command (see `retrbinary()` or a ‘LIST’ command (usually just the string ‘LIST’)). The `callback` function is called for each line, with the trailing CRLF stripped. The default `callback` prints the line to `sys.stdout`.

```
set_pasv (boolean)
```

Enable “passive” mode if `boolean` is true, other disable passive mode. (In Python 2.0 and before, passive mode was off by default; in Python 2.1 and later, it is on by default.)

```
storbinary (command, file[, blocksize])
```

Store a file in binary transfer mode. `command` should be an appropriate ‘STOR’ command, i.e. "STOR filename". `file` is an open file object which is read until EOF using its `read()` method in blocks of size `blocksize` to provide the data to be stored. The `blocksize` argument defaults to 8192. Changed in version 2.1: default for `blocksize` added.

```
storlines (command, file)
```

Store a file in ASCII transfer mode. `command` should be an appropriate ‘STOR’ command (see `storbinary()`). Lines are read until EOF from the open file object `file` using its `readline()` method to provide the data to be stored.

```
transfercmd (cmd[, rest])
```

Initiate a transfer over the data connection. If the transfer is active, send a ‘PORT’ command and the transfer command specified by `cmd`, and accept the connection. If the server is passive, send a ‘PASV’ command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional `rest` is given, a ‘REST’ command is sent to the server, passing `rest` as an argument. `rest` is usually a byte offset into the requested file, telling the server to restart sending the file’s bytes at the requested offset, skipping over the initial bytes. Note however that RFC 959 requires only that `rest` be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The `transfercmd()` method, therefore, converts `rest` to a string, but no check is performed on the string’s contents. If the server does not recognize the ‘REST’ command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a `rest` argument.

```
ntransfercmd (cmd[, rest])
```

Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. `cmd` and `rest` means the same thing as in `transfercmd()`.

```
lst (argument[, ...])
```

Return a list of files as returned by the ‘NLST’ command. The optional `argument` is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the ‘NLST’ command.

```
dir (argument[, ...])
```

Produce a directory listing as returned by the ‘LIST’ command, printing it to standard output. The optional `argument` is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the ‘LIST’ command. If the last argument is a function, it is used as a `callback` function as for `retrlines()`; the default prints to `sys.stdout`. This method returns `None`.

```
rename (fromname, toname)
```

Rename file `fromname` on the server to `toname`.

```
delete (filename)
```

Remove the file named `filename` from the server. If successful, returns the text of the response, otherwise raises

11.6. ftplib — FTP protocol client 297
error_perm on permission errors or error_reply on other errors.

**cwd**(pathname)
Set the current directory on the server.

**mkd**(pathname)
Create a new directory on the server.

**pwd()**
Return the pathname of the current directory on the server.

**rmd**(dirname)
Remove the directory named dirname on the server.

**size**(filename)
Request the size of the file named filename on the server. On success, the size of the file is returned as an integer, otherwise None is returned. Note that the ‘SIZE’ command is not standardized, but is supported by many common server implementations.

**quit()**
Send a ‘QUIT’ command to the server and close the connection. This is the “polite” way to close a connection, but it may raise an exception of the server responds with an error to the ‘QUIT’ command. This implies a call to the close() method which renders the FTP instance useless for subsequent calls (see below).

**close()**
Close the connection unilaterally. This should not be applied to an already closed connection (e.g. after a successful call to quit()). After this call the FTP instance should not be used any more (i.e., after a call to close() or quit() you cannot reopen the connection by issuing another login() method).

### 11.7 gopherlib — Gopher protocol client

This module provides a minimal implementation of client side of the the Gopher protocol. It is used by the module urllib to handle URLs that use the Gopher protocol.

The module defines the following functions:

**send_selector**(selector, host[, port])
Send a selector string to the gopher server at host and port (default 70). Returns an open file object from which the returned document can be read.

**send_query**(selector, query, host[, port])
Send a selector string and a query string to a gopher server at host and port (default 70). Returns an open file object from which the returned document can be read.

Note that the data returned by the Gopher server can be of any type, depending on the first character of the selector string. If the data is text (first character of the selector is ‘0’), lines are terminated by CRLF, and the data is terminated by a line consisting of a single ‘.’, and a leading ‘.’ should be stripped from lines that begin with ‘..’. Directory listings (first character of the selector is ‘1’) are transferred using the same protocol.

### 11.8 poplib — POP3 protocol client

This module defines a class, POP3, which encapsulates a connection to an POP3 server and implements protocol as defined in RFC 1725. The POP3 class supports both the minimal and optional command sets.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the IMAP class, as IMAP servers tend to be better implemented.
A single class is provided by the `poplib` module:

```python
class POP3 ([host[, port]])
```

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If `port` is omitted, the standard POP3 port (110) is used.

One exception is defined as an attribute of the `poplib` module:

```python
exception error_proto
```

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

### 11.8.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An `POP3` instance has the following methods:

- `getwelcome()`
  Returns the greeting string sent by the POP3 server.

- `user(username)`
  Send user command, response should indicate that a password is required.

- `pass_(password)`
  Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.

- `apop(user, secret)`
  Use the more secure APOP authentication to log into the POP3 server.

- `rpop(user)`
  Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

- `stat()`
  Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

- `list(['which'])`
  Request message list, result is in the form `(response, ['mesg_num octets', ...])`. If `which` is set, it is the message to list.

- `retr('which')`
  Retrieve whole message number `which`, and set its seen flag. Result is in form `(response, ['line', ...], octets)``

- `dele('which')`
  Flag message number `which` for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP which deliberately violates the RFCs by doing pending deletes on any disconnect).

- `rset()`
  Remove any deletion marks for the mailbox.

- `noop()`
  Do nothing. Might be used as a keep-alive.

- `quit()`
  Signoff: commit changes, unlock mailbox, drop connection.

- `top('which', howmuch)`
  Retrieves the message header plus `howmuch` lines of the message after the header of message number `which`. Result is in form `(response, ['line', ...], octets)`. 
The POP3 TOP command this method uses, unlike the RETR command, doesn’t set the message’s seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

**uidl** ([`which`])

Return message digest (unique id) list. If `which` is specified, result contains the unique id for that message in the form `response mesgnum uid`, otherwise result is list (`response`, `[`mesgnum uid`, ...`], `octets`).

**See Also:**

Module `imap` (section ??):
The standard Python IMAP module.

`http://www.tuxedo.org/esr/fetchail/fetchmail-FAQ.html`
The FAQ for the fetchmail POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on poplib.

### 11.8.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

At the end of the module, there is a test section that contains a more extensive example of usage.

### 11.9 imaplib — IMAP4 protocol client

This module defines a class, `IMAP4`, which encapsulates a connection to an IMAP4 server and implements a large subset of the IMAP4rev1 client protocol as defined in RFC 2060. It is backward compatible with IMAP4 (RFC 1730) servers, but note that the ‘STATUS’ command is not supported in IMAP4.

A single class is provided by the `imaplib` module:

**class IMAP4 ([`host`, `port`])**

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If `host` is not specified, ‘’ (the local host) is used. If `port` is omitted, the standard IMAP4 port (143) is used.

Two exceptions are defined as attributes of the `IMAP4` class:

**exception IMAP4.error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

**exception IMAP4.abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of `IMAP4.error`. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

**exception IMAP4readonly**
This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of \texttt{IMAP4.error}. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

The following utility functions are defined:

\textbf{Internaldate2tuple} (\texttt{datestr})

Converts an IMAP4 \texttt{INTERNALDATE} string to Coordinated Universal Time. Returns a \texttt{time} module tuple.

\textbf{Int2AP} (\texttt{num})

Converts an integer into a string representation using characters from the set [A..P].

\textbf{ParseFlags} (\texttt{flagstr})

Converts an IMAP4 ‘FLAGS’ response to a tuple of individual flags.

\textbf{Time2Internaldate} (\texttt{date\_time})

Converts a \texttt{time} module tuple to an IMAP4 ‘INTERNALDATE’ representation. Returns a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes).

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an ‘EXPUNGE’ command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

See Also:
Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington’s IMAP Information Center (http://www.cac.washington.edu/imap/).

11.9.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for ‘AUTHENTICATE’, and the last argument to ‘APPEND’ which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn’t enclosed with either parentheses or double quotes) each string is quoted. However, the \texttt{password} argument to the ‘LOGIN’ command is always quoted. If you want to avoid having an argument string quoted (eg: the \texttt{flags} argument to ‘STORE’) then enclose the string in parentheses (eg: \texttt{r’(\Deleted)’}).

Each command returns a tuple: \texttt{(type, [data, ...])} where \texttt{type} is usually ‘\texttt{OK}’ or ‘\texttt{NO}’, and \texttt{data} is either the text from the command response, or mandated results from the command.

An IMAP4 instance has the following methods:

\textbf{append} (\texttt{mailbox, flags, date\_time, message})

Append message to named mailbox.

\textbf{authenticate} (\texttt{func})

Authenticate command — requires response processing. This is currently unimplemented, and raises an exception.

\textbf{check} ()

Checkpoint mailbox on server.

\textbf{close} ()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before ‘LOGOUT’.

\textbf{copy} (\texttt{message\_set, new\_mailbox})

Copy \texttt{message\_set} messages onto end of \texttt{new\_mailbox}.

\textbf{create} (\texttt{mailbox})
Create new mailbox named `mailbox`.

`delete (mailbox)`
Delete old mailbox named `mailbox`.

`expunge ()`
Permanently remove deleted items from selected mailbox. Generates an ‘EXPUNGE’ response for each deleted message. Returned data contains a list of ‘EXPUNGE’ message numbers in order received.

`fetch (message_set, message_parts)`
Fetch (parts of) messages. `message_parts` should be a string of message part names enclosed within parentheses, eg: ‘" (UID BODY[TEXT])"’. Returned data are tuples of message part envelope and data.

`list ([directory[, pattern ]])`
List mailbox names in `directory` matching `pattern`. `directory` defaults to the top-level mail folder, and `pattern` defaults to match anything. Returned data contains a list of ‘LIST’ responses.

`login (user, password)`
Identify the client using a plaintext password. The `password` will be quoted.

`logout ()`
Shutdown connection to server. Returns server ‘BYE’ response.

`lsub ([directory[, pattern ]])`
List subscribed mailbox names in directory matching pattern. `directory` defaults to the top level directory and `pattern` defaults to match any mailbox. Returned data are tuples of message part envelope and data.

`noop ()`
Send ‘NOOP’ to server.

`open (host, port)`
Opens socket to `port` at `host`. You may override this method.

`partial (message_num, message_part, start, length)`
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

`recent ()`
Prompt server for an update. Returned data is `None` if no new messages, else value of ‘RECENT’ response.

`rename (oldmailbox, newmailbox)`
Rename mailbox named `oldmailbox` to `newmailbox`.

`response (code)`
Return data for response `code` if received, or `None`. Returns the given code, instead of the usual type.

`search (charset, criterium[...])`
Search mailbox for matching messages. Returned data contains a space separated list of matching message numbers. `charset` may be `None`, in which case no ‘CHARSET’ will be specified in the request to the server. The IMAP protocol requires that at least one criterium be specified; an exception will be raised when the server returns an error.

Example:

```python
# M is a connected IMAP4 instance...
msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
msgnums = M.search(None, '(FROM "LDJ")')
```

`select ([mailbox[, readonly ]])`
Select a mailbox. Returned data is the count of messages in `mailbox` (‘EXISTS’ response). The default `mailbox` is ‘INBOX’. If the `readonly` flag is set, modifications to the mailbox are not allowed.
socket()
   Returns socket instance used to connect to server.

status(mailbox, names)
   Request named status conditions for mailbox.

store(message_set, command, flag_list)
   Alters flag dispositions for messages in mailbox.

subscribe(mailbox)
   Subscribe to new mailbox.

uid(command, arg[...])
   Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

unsubscribe(mailbox)
   Unsubscribe from old mailbox.

xatom(name[, arg[, ...]]
   Allow simple extension commands notified by server in ‘CAPABILITY’ response.

The following attributes are defined on instances of IMAP4:

PROTOCOL_VERSION
   The most recent supported protocol in the ‘CAPABILITY’ response from the server.

debug
   Integer value to control debugging output. The initialize value is taken from the module variable Debug. Values greater than three trace each command.

11.9.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, imaplib, string

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()

for num in string.split(data[0]):
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s
%s
' % (num, data[0][1])

M.logout()
```

11.10 nntplib — NNTP protocol client

This module defines the class NNTP which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:
>>> s = NNTP('news.cwi.nl')
>>> resp, count, first, last, name = s.group('comp.lang.python')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
Group comp.lang.python has 59 articles, range 3742 to 3803
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
... 3792 Re: Removing elements from a list while iterating...
3793 Re: Who likes Info files?
3794 Emacs and doc strings
3795 a few questions about the Mac implementation
3796 Re: executable python scripts
3797 Re: executable python scripts
3798 Re: a few questions about the Mac implementation
3799 Re: PROPOSAL: A Generic Python Object Interface for Python C Modules
3802 Re: executable python scripts
3803 Re: \
POSIX{} wait and SIGCHLD
>>> s.quit()
'205 news.cwi.nl closing connection. Goodbye.'

To post an article from a file (this assumes that the article has valid headers):

>>> s = NNTP('news.cwi.nl')
>>> f = open('/tmp/article')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 news.cwi.nl closing connection. Goodbye.'

The module itself defines the following items:

class NNTP (host[, port [, user[, password [, readermode ]]]]])
Return a new instance of the NNTP class, representing a connection to the NNTP server running on host host,
listening at port port. The default port is 119. If the optional user and password are provided, the 'AUTHINFO
USER' and 'AUTHINFO PASS' commands are used to identify and authenticate the user to the server. If the
optional flag readermode is true, then a 'mode reader' command is sent before authentication is performed.
Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend
to call reader-specific commands, such as 'group'. If you get unexpected NNTPPermanentErrors, you
might need to set readermode. readermode defaults to None.

class NNTPError ()
Derived from the standard exception Exception, this is the base class for all exceptions raised by the
ntplib module.

class NNTPReplyError ()
Exception raised when an unexpected reply is received from the server. For backwards compatibility, the exception error_reply is equivalent to this class.

class NNTPTemporaryError ()
Exception raised when an error code in the range 400–499 is received. For backwards compatibility, the exception error_temp is equivalent to this class.

class NNTPPermanentError ()
Exception raised when an error code in the range 500–599 is received. For backwards compatibility, the exception error_perm is equivalent to this class.
class NNTPProtocolError()
   Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5. For
   backwards compatibility, the exception error_proto is equivalent to this class.

class NNTPDataError()
   Exception raised when there is some error in the response data. For backwards compatibility, the exception
   error_data is equivalent to this class.

11.10.1 NNTP Objects

NNTP instances have the following methods. The response that is returned as the first item in the return tuple of almost
all methods is the server’s response: a string beginning with a three-digit code. If the server’s response indicates an
error, the method raises one of the above exceptions.

getwelcome()
   Return the welcome message sent by the server in reply to the initial connection. (This message sometimes
   contains disclaimers or help information that may be relevant to the user.)

set_debuglevel(level)
   Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

newgroups(date, time)
   Send a ‘NEWGROUPS’ command. The date argument should be a string of the form ’yyyymmdd’ indicating the
date, and time should be a string of the form ’hhmmss’ indicating the time. Return a pair (response, groups) where groups is a list of group names that are new since the given date and time.

newnews(group, date, time)
   Send a ‘NEWNEWS’ command. Here, group is a group name or ‘*’, and date and time have the same meaning
   as for newgroups(). Return a pair (response, articles) where articles is a list of article ids.

list()
   Send a ‘LIST’ command. Return a pair (response, list) where list is a list of tuples. Each tuple has the form
   (group, last, first, flag), where group is a group name, last and first are the last and first article numbers
   (as strings), and flag is ’y’ if posting is allowed, ’n’ if not, and ’m’ if the newsgroup is moderated. (Note the
   ordering: last, first.)

group(name)
   Send a ‘GROUP’ command, where name is the group name. Return a tuple (response, count, first, last,
   name) where count is the (estimated) number of articles in the group, first is the first article number in the
   group, last is the last article number in the group, and name is the group name. The numbers are returned as
   strings.

help()
   Send a ‘HELP’ command. Return a pair (response, list) where list is a list of help strings.

stat(id)
   Send a ‘STAT’ command, where id is the message id (enclosed in ‘<’ and ‘>’) or an article number (as a string).
   Return a triple (response, number, id) where number is the article number (as a string) and id is the article
   id (enclosed in ‘<’ and ‘>’).

next()
   Send a ‘NEXT’ command. Return as for stat().

last()
   Send a ‘LAST’ command. Return as for stat().
head(id)
Send a ‘HEAD’ command, where id has the same meaning as for stat(). Return a tuple (response, number, id, list) where the first three are the same as for stat(), and list is a list of the article’s headers (an uninterpreted list of lines, without trailing newlines).

body(id)
Send a ‘BODY’ command, where id has the same meaning as for stat(). Return as for head().

article(id)
Send an ‘ARTICLE’ command, where id has the same meaning as for stat(). Return as for head().

slave()
Send a ‘SLAVE’ command. Return the server’s response.

xhdr (header, string)
Send an ‘XHDR’ command. This command is not defined in the RFC but is a common extension. The header argument is a header keyword, e.g. ‘subject’. The string argument should have the form ‘first–last’ where first and last are the first and last article numbers to search. Return a pair (response, list), where list is a list of pairs (id, text), where id is an article id (as a string) and text is the text of the requested header for that article.

post (file)
Post an article using the ‘POST’ command. The file argument is an open file object which is read until EOF using its readline() method. It should be a well-formed news article, including the required headers. The post() method automatically escapes lines beginning with ‘.’.

ihave (id, file)
Send an ‘IHAVE’ command. If the response is not an error, treat file exactly as for the post() method.

date()
Return a triple (response, date, time), containing the current date and time in a form suitable for the newnews() and newgroups() methods. This is an optional NNTP extension, and may not be supported by all servers.

xgtitle (name)
Process an ‘XGTITLE’ command, returning a pair (response, list), where list is a list of tuples containing (name, title). This is an optional NNTP extension, and may not be supported by all servers.

xover (start, end)
Return a pair (resp, list). list is a list of tuples, one for each article in the range delimited by the start and end article numbers. Each tuple is of the form (article number, subject, poster, date, id, references, size, lines). This is an optional NNTP extension, and may not be supported by all servers.

xpath (id)
Return a pair (resp, path), where path is the directory path to the article with message ID id. This is an optional NNTP extension, and may not be supported by all servers.

quit()
Send a ‘QUIT’ command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

11.11 smtplib — SMTP protocol client

The smtplib module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult RFC 821 (Simple Mail Transfer Protocol) and RFC 1869 (SMTP Service Extensions).

class SMTP ([host[, port]])
A SMTP instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and
ESMTP operations. If the optional host and port parameters are given, the SMTP `connect()` method is called with those parameters during initialization. An `SMTPConnectError` is raised if the specified host doesn’t respond correctly.

For normal use, you should only require the initialization/connect, `sendmail()`, and `quit()` methods. An example is included below.

A nice selection of exceptions is defined as well:

**exception `SMTPException`**
Base exception class for all exceptions raised by this module.

**exception `SMTPServerDisconnected`**
This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the SMTP instance before connecting it to a server.

**exception `SMTPResponseException`**
Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

**exception `SMTPSenderRefused`**
Sender address refused. In addition to the attributes set by all `SMTPResponseException` exceptions, this sets ‘sender’ to the string that the SMTP server refused.

**exception `SMTPRecipientsRefused`**
All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as SMTP.`sendmail()` returns.

**exception `SMTPDataError`**
The SMTP server refused to accept the message data.

**exception `SMTPConnectError`**
Error occurred during establishment of a connection with the server.

**exception `SMTPHeloError`**
The server refused our ‘HELO’ message.

See Also:

RFC 821, “Simple Mail Transfer Protocol”
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869, “SMTP Service Extensions”
Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

### 11.11.1 SMTP Objects

An SMTP instance has the following methods:

**`set_debuglevel(level)`**
Set the debug output level. A true value for `level` results in debug messages for connection and for all messages sent to and received from the server.

**`connect([host[, port]])`**
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25).
If the hostname ends with a colon (‘:’) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use.

---

11.11. smtplib — SMTP protocol client
Note: This method is automatically invoked by the constructor if a host is specified during instantiation.

**docmd**(cmd, [, argstring])

Send a command `cmd` to the server. The optional argument `argstring` is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, **SMTPServerDisconnected** will be raised.

**helo**(hostname)

Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the **sendmail()** when necessary.

**ehlo**(hostname)

Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by **has_option()**.

Unless you wish to use **has_option()** before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by **sendmail()** when necessary.

**has_extn**(name)

Return 1 if name is in the set of SMTP service extensions returned by the server, 0 otherwise. Case is ignored.

**verify**(address)

Check the validity of an address on this server using SMTP `VRFY`. Returns a tuple consisting of code 250 and a full RFC 822 address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

Note: many sites disable SMTP `VRFY` in order to foil spammers.

**sendmail**(from_addr, to_addrs, msg[, mail_options, rcpt_options])

Send mail. The required arguments are an RFC 822 from-address string, a list of RFC 822 to-address strings, and a message string. The caller may pass a list of ESMTP options (such as `8bitmime`) to be used in `MAIL FROM` commands as mail_options. ESMTP options (such as `DSN` commands) that should be used with all `RCPT` commands can be passed as rcpt_options. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as mail, rcpt and data to send the message.)

**Note:** The `from_addr` and `to_addrs` parameters are used to construct the message envelope used by the transport agents. The SMTP does not modify the message headers in any way.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first.

If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will throw an exception. That is, if this method does not throw an exception, then someone should get your mail. If this method does not throw an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

This method may raise the following exceptions:

**SMTPRecipientsRefused** All recipients were refused. Nobody got the mail. The recipients attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

**SMTPHeloError** The server didn’t reply properly to the `HELO` greeting.
SMTPSenderRefused: The server didn’t accept the from_addr.
SMTPDataError: The server replied with an unexpected error code (other than a refusal of a recipient).

Unless otherwise noted, the connection will be open even after an exception is raised.

quit()
Terminate the SMTP session and close the connection.

Low-level methods corresponding to the standard SMTP/ESMTP commands ‘HELP’, ‘RSET’, ‘NOOP’, ‘MAIL’, ‘RCPT’, and ‘DATA’ are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

11.11.2 SMTP Example

This example prompts the user for addresses needed in the message envelope (‘To’ and ‘From’ addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn’t do any processing of the RFC 822 headers. In particular, the ‘To’ and ‘From’ addresses must be included in the message headers explicitly.

```python
import smtplib
import string

def prompt(prompt):
    return string.strip(raw_input(prompt))

fromaddr = prompt("From: ")
toaddrs = string.split(prompt("To: "))
print "Enter message, end with \D:"

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\n" % (fromaddr, string.join(toaddrs, ", "))
while 1:
    line = raw_input()
    if not line:
        break
    msg = msg + line

print "Message length is " + 'len(msg)'

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

11.12 telnetlib — Telnet client

The telnetlib module provides a Telnet class that implements the Telnet protocol. See RFC 854 for details about the protocol.

class Telnet ([host[, port]])
Telnet represents a connection to a telnet server. The instance is initially not connected by default; the
The `open()` method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor returns.

Do not reopen an already connected instance.

This class has many read_*() methods. Note that some of them raise EOFError when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

See Also:
Definition of the Telnet protocol.

### 11.12.1 Telnet Objects

Telnet instances have the following methods:

**read_until(expected[, timeout])**

Read until a given string is encountered or until timeout.

When no match is found, return whatever is available instead, possibly the empty string. Raise EOFError if the connection is closed and no cooked data is available.

**read_all()**

Read all data until EOF; block until connection closed.

**read_some()**

Read at least one byte of cooked data unless EOF is hit. Return '' if EOF is hit. Block if no data is immediately available.

**read_very_eager()**

Read everything that can be without blocking in I/O (eager).

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

**read_eager()**

Read readily available data.

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

**read_lazy()**

Process and return data already in the queues (lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

**read_very_lazy()**

Return any data available in the cooked queue (very lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. This method never blocks.

**open(host[, port])**

Connect to a host. The optional second argument is the port number, which defaults to the standard telnet port (23).

Do not try to reopen an already connected instance.

**msg(msg[, *args])**

Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.
**set_debuglevel (debuglevel)**
Set the debug level. The higher the value of debuglevel, the more debug output you get (on sys.stdout).

**close ()**
Close the connection.

**get_socket ()**
Return the socket object used internally.

**fileno ()**
Return the file descriptor of the socket object used internally.

**write (buffer)**
Write a string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise socket.error if the connection is closed.

**interact ()**
Interaction function, emulates a very dumb telnet client.

**mt_interact ()**
Multithreaded version of interact().

**expect (list[, timeout])**
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (re.RegexObject instances) or uncompiled (strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.

If end of file is found and no text was read, raise EOFError. Otherwise, when nothing matches, return (-1, None, text) where text is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (e.g. ⌈ .* ⌋) or if more than one expression can match the same input, the results are indeterministic, and may depend on the I/O timing.

### 11.12.2 Telnet Example

A simple example illustrating typical use:
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)
	n.read_until("login: ")

tn.write(user + "\n")
if password:
    tn.read_until("Password: ")
    tn.write(password + "\n")

tn.write("ls\n")
tn.write("exit\n")

print tn.read_all()

11.13 urlparse — Parse URLs into components

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators (and discovered a bug in an earlier draft!).

It defines the following functions:

urlparse(urlstring[, default_scheme[, allow_fragments]])

Parse a URL into 6 components, returning a 6-tuple: (addressing scheme, network location, path, parameters, query, fragment identifier). This corresponds to the general structure of a URL: scheme://netloc/path;parameters?query#fragment. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (e.g. the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the tuple items, except for a leading slash in the path component, which is retained if present.

Example:

urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')

yields the tuple

('http', 'www.cwi.nl:80', '%7Eguido/Python.html', '', '', '')

If the default_scheme argument is specified, it gives the default addressing scheme, to be used only if the URL string does not specify one. The default value for this argument is the empty string.

If the allow_fragments argument is zero, fragment identifiers are not allowed, even if the URL's addressing scheme normally does support them. The default value for this argument is 1.
urlunparse (tuple)

Construct a URL string from a tuple as returned by urlparse(). This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had redundant delimiters, e.g. a ? with an empty query (the draft states that these are equivalent).

urljoin (base, url[, allow_fragments ])

Construct a full (“absolute”) URL by combining a “base URL” (base) with a “relative URL” (url). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL.

Example:

    urljoin(’http://www.cwi.nl/%7Eguido/Python.html’, ’FAQ.html’)

yields the string

    ’http://www.cwi.nl/%7Eguido/FAQ.html’

The allow_fragments argument has the same meaning as for urlparse().

See Also:

RFC 1738, “Uniform Resource Locators (URL)”
This specifies the formal syntax and semantics of absolute URLs.

RFC 1808, “Relative Uniform Resource Locators”
This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair normal of “Abnormal Examples” which govern the treatment of border cases.

RFC 2396, “Uniform Resource Identifiers (URI): Generic Syntax”
Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

11.14 SocketServer — A framework for network servers

The SocketServer module simplifies the task of writing network servers.

There are four basic server classes: TCPServer uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. UDPServer uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The more infrequently used UnixStreamServer and UnixDatagramServer classes are similar, but use UNIX domain sockets; they’re not available on non-UNIX platforms. For more details on network programming, consult a book such as W. Richard Steven’s UNIX Network Programming or Ralph Davis’s Win32 Network Programming.

These four classes process requests synchronously; each request must be completed before the next request can be started. This isn’t suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the BaseRequestHandler class and overriding its handle() method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server’s address and the request handler class. Finally, call the handle_request() or serve_forever() method of the server object to process one or many requests.

Server classes have the same external methods and attributes, no matter what network protocol they use:
Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to `select.select()`, to allow monitoring multiple servers in the same process.

Process a single request. This function calls the following methods in order: `get_request()`, `verify_request()`, and `process_request()`. If the user-provided `handle()` method of the handler class raises an exception, the server's `handle_error()` method will be called.

Handle an infinite number of requests. This simply calls `handle_request()` inside an infinite loop.

The family of protocols to which the server’s socket belongs. `socket.AF_INET` and `socket.AF_UNIX` are two possible values.

The user-provided request handler class; an instance of this class is created for each request.

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the socket module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: (`'127.0.0.1'`, `80`), for example.

The socket object on which the server will listen for incoming requests.

Whether the server will allow the reuse of an address. This defaults to true, and can be set in subclasses to change the policy.

The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue is full, further requests from clients will get a “Connection denied” error. The default value is usually 5, but this can be overridden by subclasses.

The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two possible values.

There are various server methods that can be overridden by subclasses of base server classes like `TCPServer`; these methods aren’t useful to external users of the server object.

Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()` method.

Must accept a request from the socket, and return a 2-tuple containing the new socket object to be used to communicate with the client, and the client’s address.

This function is called if the `RequestHandlerClass`'s `handle()` method raises an exception. The default action is to print the traceback to standard output and continue handling further requests.

Calls `finish_request()` to create an instance of the `RequestHandlerClass`. If desired, this function can create a new process or thread to handle the request; the `ForkingMixIn` and `ThreadingMixIn` classes do this.
server_activate()
 Called by the server’s constructor to activate the server. May be overridden.

server_bind()
 Called by the server’s constructor to bind the socket to the desired address. May be overridden.

verify_request (request, client_address)
 Must return a Boolean value; if the value is true, the request will be processed, and if it’s false, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always return true.

The request handler class must define a new handle() method, and can override any of the following methods. A new instance is created for each request.

finish()
 Called after the handle() method to perform any clean-up actions required. The default implementation does nothing. If setup() or handle() raise an exception, this function will not be called.

handle()
 This function must do all the work required to service a request. Several instance attributes are available to it; the request is available as self.request; the client address as self.client_address; and the server instance as self.server, in case it needs access to per-server information.

The type of self.request is different for datagram or stream services. For stream services, self.request is a socket object; for datagram services, self.request is a string. However, this can be hidden by using the mix-in request handler classes StreamRequestHandler or DatagramRequestHandler, which override the setup() and finish() methods, and provides self.rfile and self.wfile attributes. self.rfile and self.wfile can be read or written, respectively, to get the request data or return data to the client.

setup()
 Called before the handle() method to perform any initialization actions required. The default implementation does nothing.

11.15 BaseHTTPServer — Basic HTTP server

This module defines two classes for implementing HTTP servers (web servers). Usually, this module isn’t used directly, but is used as a basis for building functioning web servers. See the SimpleHTTPServer and CGIHTTPServer modules.

The first class, HTTPServer, is a SocketServer.TCPServer subclass. It creates and listens at the web socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```python
def run(server_class=BaseHTTPServer.HTTPServer,
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class HTTPServer (server_address, RequestHandlerClass)

This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler’s server instance variable.

class BaseHTTPRequestHandler (request, client_address, server)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to
any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method 'SPAM', the do_SPAM() method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the __init__() method.

BaseHTTPRequestHandler has the following instance variables:

client_address
- Contains a tuple of the form (host, port) referring to the client’s address.

command
- Contains the command (request type). For example, 'GET'.

path
- Contains the request path.

request_version
- Contains the version string from the request. For example, 'HTTP/1.0'.

headers
- Holds an instance of the class specified by the MessageClass class variable. This instance parses and manages the headers in the HTTP request.

rfile
- Contains an input stream, positioned at the start of the optional input data.

wfile
- Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream.

BaseHTTPRequestHandler has the following class variables:

server_version
- Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, 'BaseHTTP/0.2'.

sys_version
- Contains the Python system version, in a form usable by the version_string method and the server_version class variable. For example, 'Python/1.4'.

error_message_format
- Specifies a format string for building an error response to the client. It uses parenthesized, keyed format specifiers, so the format operand must be a dictionary. The code key should be an integer, specifying the numeric HTTP error code value. message should be a string containing a (detailed) error message of what occurred, and explain should be an explanation of the error code number. Default message and explain values can found in the responses class variable.

protocol_version
- This specifies the HTTP protocol version used in responses. Typically, this should not be overridden. Defaults to 'HTTP/1.0'.

MessageClass
- Specifies a rfc822.Message-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to mimetools.Message.

responses
- This variable contains a mapping of error code integers to two-element tuples containing a short and long message. For example, (code: (shortmessage, longmessage)). The shortmessage is usually used as the message key in an error response, and longmessage as the explain key (see the error_message_format
A `BaseHTTPRequestHandler` instance has the following methods:

**handle()**
Overides the superclass’ `handle()` method to provide the specific handler behavior. This method will parse and dispatch the request to the appropriate `do_*()` method.

**send_error(code[, message])**
Sends and logs a complete error reply to the client. The numeric `code` specifies the HTTP error code, with `message` as optional, more specific text. A complete set of headers is sent, followed by text composed using the `error_message_format` class variable.

**send_response(code[, message])**
Sends a response header and logs the accepted request. The HTTP response line is sent, followed by `Server` and `Date` headers. The values for these two headers are picked up from the `version_string()` and `date_time_string()` methods, respectively.

**send_header(keyword, value)**
Writes a specific MIME header to the output stream. `keyword` should specify the header keyword, with `value` specifying its value.

**end_headers()**
Sends a blank line, indicating the end of the MIME headers in the response.

**log_request([code[, size]])**
Logs an accepted (successful) request. `code` should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the `size` parameter.

**log_error(...)**
Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (`format` and additional values).

**log_message(format, ...)**
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The `format` argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client address and current date and time are prefixed to every message logged.

**version_string()**
Returns the server software’s version string. This is a combination of the `server_version` and `sys_version` class variables.

**date_time_string()**
Returns the current date and time, formatted for a message header.

**log_data_time_string()**
Returns the current date and time, formatted for logging.

**address_string()**
Returns the client address, formatted for logging. A name lookup is performed on the client’s IP address.

See Also:

Module `CGIHTTPServer` (section 11.17):
Extended request handler that supports CGI scripts.

Module `SimpleHTTPServer` (section 11.16):
Basic request handler that limits response to files actually under the document root.
11.16 SimpleHTTPServer — Simple HTTP request handler

The SimpleHTTPServer module defines a request-handler class, interface compatible with BaseHTTPServer.BaseHTTPRequestHandler which serves files only from a base directory.

The SimpleHTTPServer module defines the following class:

class SimpleHTTPRequestHandler (request, client_address, server)

This class is used, to serve files from current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work is done by the base class BaseHTTPServer.BaseHTTPRequestHandler, such as parsing the request. This class implements the do_GET() and do_HEAD() functions.

The SimpleHTTPRequestHandler defines the following member variables:

server_version
This will be "SimpleHTTP/" + __version__, where __version__ is defined in the module.

extensions_map
A dictionary mapping suffixes into MIME types. Default is signified by an empty string, and is considered to be text/plain. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The SimpleHTTPRequestHandler defines the following methods:

do_HEAD ()
This method serves the ‘HEAD’ request type: it sends the headers it would send for the equivalent GET request. See the do_GET () method for more complete explanation of the possible headers.

do_GET ()
The request is mapped to a local file by interpreting the request as a path relative to the current working directory. If the request was mapped to a directory, a 403 respond is output, followed by the explanation ‘Directory listing not supported’. Any IOError exception in opening the requested file, is mapped to a 404, ‘File not found’ error. Otherwise, the content type is guessed using the extensions_map variable.

A ‘Content-type:’ with the guessed content type is output, and then a blank line, signifying end of headers, and then the contents of the file. The file is always opened in binary mode.

For example usage, see the implementation of the test() function.

See Also:
Module BaseHTTPServer (section 11.15):
Base class implementation for Web server and request handler.

11.17 CGIHTTPServer — CGI-capable HTTP request handler

The CGIHTTPServer module defines a request-handler class, interface compatible with BaseHTTPServer.BaseHTTPRequestHandler and inherits behavior from SimpleHTTPServer.SimpleHTTPRequestHandler but can also run CGI scripts.

Note: This module is UNIX dependent since it creates the CGI process using os.fork() and os.exec().

The CGIHTTPServer module defines the following class:

class CGIHTTPRequestHandler (request, client_address, server)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in SimpleHTTPServer.SimpleHTTPRequestHandler.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only
directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The `CGIHTTPRequestHandler` defines the following data member:

**cgi_directories**
This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The `CGIHTTPRequestHandler` defines the following methods:

**do_POST()**
This method serves the `POST` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

For example usage, see the implementation of the `test()` function.

**See Also:**
Module `BaseHTTPServer` (section 11.15):
Base class implementation for Web server and request handler.

### 11.18 Cookie — HTTP state management

The `Cookie` module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in in the RFC 2109 and RFC 2068 specifications. It has since been discovered that MSIE 3.0x doesn’t follow the character rules outlined in those specs. As a result, the parsing rules used are a bit less strict.

**exception CookieError**
Exception failing because of RFC 2109 invalidity: incorrect attributes, incorrect `Set-Cookie` header, etc.

**class BaseCookie ([input])**
This class is a dictionary-like object whose keys are strings and whose values are `Morsel` s. Note that upon setting a key to a value, the value is first converted to a `Morsel` containing the key and the value.

If `input` is given, it is passed to the `load` method.

**class SimpleCookie ([input])**
This class derives from `BaseCookie` and overrides `value_decode` and `value_encode` to be the identity and `str()` respectively.

**class SerialCookie ([input])**
This class derives from `BaseCookie` and overrides `value_decode` and `value_encode` to be the `pickle.loads()` and `pickle.dumps()`.

Do not use this class. Reading pickled values from a cookie is a security hole, as arbitrary client-code can be run on `pickle.loads()`. It is supported for backwards compatibility.

**class SmartCookie ([input])**
This class derives from `BaseCookie`. It overrides `value_decode` to be `pickle.loads()` if it is a valid pickle, and otherwise the value itself. It overrides `value_encode` to be `pickle.dumps()` unless it is a string, in which case it returns the value itself.

The same security warning from `SerialCookie` applies here.
See Also:

RFC 2109, “HTTP State Management Mechanism”
    This is the state management specification implemented by this module.

11.18.1 Cookie Objects

value_decode(val)
    Return a decoded value from a string representation. Return value can be any type. This method does nothing in BaseCookie — it exists so it can be overridden.

value_encode(val)
    Return an encoded value. val can be any type, but return value must be a string. This method does nothing in BaseCookie — it exists so it can be overridden.
    In general, it should be the case that value_encode and value_decode are inverses on the range of value_decode.

output(attrs, header, sep)
    Return a string representation suitable to be sent as HTTP headers. attrs and header are sent to each Morsel’s output method. sep is used to join the headers together, and is by default a newline.

js_output(attrs)
    Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.
    The meaning for attrs is the same as in output().

load(rawdata)
    If rawdata is a string, parse it as an HTTP_COOKIE and add the values found there as Morsels. If it is a dictionary, it is equivalent to:

        for k, v in rawdata.items():
            cookie[k] = v

11.18.2 Morsel Objects

class Morsel()
    Abstract a key/value pair, which has some RFC 2109 attributes.
    Morsels are dictionary-like objects, whose set of keys is constant — the valid RFC 2109 attributes, which are

    • expires
    • path
    • comment
    • domain
    • max-age
    • secure
    • version

    The keys are case-insensitive.

value
    The value of the cookie.
The encoded value of the cookie — this is what should be sent.

The name of the cookie.

Set the key, value and coded_value members.

Whether K is a member of the set of keys of a Morsel.

Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless attrs is given, in which case it should be a list of attributes to use. header is by default "Set-Cookie:"

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for attrs is the same as in output().

Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for attrs is the same as in output().

The following example demonstrates how to use the Cookie module.
>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C = Cookie.SerialCookie()
>>> C = Cookie.SmartCookie()
>>> C = Cookie.Cookie() # backwards-compatible alias for SmartCookie
>>> C = Cookie.SmartCookie()
>>> C['fig'] = "newton"
>>> C['sugar'] = "wafer"
>>> print C  # generate HTTP headers
Set-Cookie: sugar=wafer;
Set-Cookie: fig=newton;
>>> print C.output()  # same thing
Set-Cookie: sugar=wafer;
Set-Cookie: fig=newton;
>>> C = Cookie.SmartCookie()
>>> C['rocky'] = "road"
>>> C['rocky']['path'] = "/cookie"
>>> print C.output(header="Cookie: ")
Cookie: rocky=road; Path=/cookie;
>>> print C.output(attrs=[], header="Cookie: ")
Cookie: rocky=road;
>>> C = Cookie.SmartCookie()
>>> C.load("chips=ahoy; vienna=finger")  # load from a string (HTTP header)
>>> print C
Set-Cookie: vienna=finger;
Set-Cookie: chips=ahoy;
>>> C = Cookie.SmartCookie()
>>> C.load('keebler="E=everybody; L="Loves\"; fudge=\012;";')
>>> print C
Set-Cookie: keebler="E=everybody; L="Loves\"; fudge=\012;";
>>> C = Cookie.SmartCookie()
>>> C['oreo'] = "doublestuff"
>>> C['oreo']['path'] = "/"
>>> print C
Set-Cookie: oreo=doublestuff; Path=/;
>>> C = Cookie.SmartCookie()
>>> C['twix'] = "none for you"
>>> C['twix'].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C['number'] = 7 # equivalent to C['number'] = str(7)
>>> C['string'] = "seven"
>>> C['number'].value
'7'
>>> C['string'].value
'seven'
>>> print C
Set-Cookie: number=7;
Set-Cookie: string=seven;
>>> C = Cookie.SerialCookie()
>>> C['number'] = 7
>>> C['string'] = "seven"
>>> C['number'].value
7
>>> C['string'].value
'seven'
>>> print C
Set-Cookie: number="I7\012.";
Set-Cookie: string="S'seven'\012p1\012.";
>>> C = Cookie.SmartCookie()
11.19 asyncore — Asynchronous socket handler

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do “more than one thing at a time.” Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It’s really only practical if your program is largely I/O bound. If your program is CPU bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely CPU-bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the “background.” Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The module documented here solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap.

class dispatcher()

The first class we will introduce is the `dispatcher` class. This is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling on it. Otherwise, it can be treated as a normal non-blocking socket object.

The direct interface between the select loop and the socket object are the `handle_read_event()` and `handle_write_event()` methods. These are called whenever an object ‘fires’ that event.

The firing of these low-level events can tell us whether certain higher-level events have taken place, depending on the timing and the state of the connection. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket fires a write event (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_connect()</td>
<td>Implied by a write event</td>
</tr>
<tr>
<td>handle_close()</td>
<td>Implied by a read event with no data available</td>
</tr>
<tr>
<td>handle_accept()</td>
<td>Implied by a read event on a listening socket</td>
</tr>
</tbody>
</table>

This set of user-level events is larger than the basics. The full set of methods that can be overridden in your subclass are:

`handle_read()`

Called when there is new data to be read from a socket.

`handle_write()`

Called when there is an attempt to write data to the object. Often this method will implement the necessary buffering for performance. For example:

```python
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

`handle_expt()`

Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

`handle_connect()`

Called when the socket actually makes a connection. This might be used to send a “welcome” banner, or something similar.

`handle_close()`

Called when the socket is closed.

`handle_accept()`

Called on listening sockets when they actually accept a new connection.
**readable()**
Each time through the `select()` loop, the set of sockets is scanned, and this method is called to see if there is any interest in reading. The default method simply returns 1, indicating that by default, all channels will be interested.

**writable()**
Each time through the `select()` loop, the set of sockets is scanned, and this method is called to see if there is any interest in writing. The default method simply returns 1, indicating that by default, all channels will be interested.

In addition, there are the basic methods needed to construct and manipulate “channels,” which are what we will call the socket connections in this context. Note that most of these are nearly identical to their socket partners.

**create_socket (family, type)**
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the `socket` documentation for information on creating sockets.

**connect (address)**
As with the normal socket object, `address` is a tuple with the first element the host to connect to, and the second the port.

**send (data)**
Send `data` out the socket.

**recv (buffer_size)**
Read at most `buffer_size` bytes from the socket.

**listen ([backlog])**
Listen for connections made to the socket. The `backlog` argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

**bind (address)**
Bind the socket to `address`. The socket must not already be bound. (The format of `address` depends on the address family — see above.)

**accept ()**
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (`conn`, `address`) where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

**close ()**
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

### 11.19.1 Example basic HTTP client

As a basic example, below is a very basic HTTP client that uses the `dispatcher` class to implement its socket handling:

```python
# Example basic HTTP client
import socket

# Create a socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to a server
sock.connect(('example.com', 80))

# Send some data
sock.send(b'GET / HTTP/1.0\r\n\r\n')

# Receive the response
response = sock.recv(4096)

# Close the connection
sock.close()
```
class http_client(asyncore.dispatcher):
    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.path = path
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect((host, 80))
        self.buffer = 'GET %s HTTP/1.0\r\n\n' % self.path

    def handle_connect(self):
        pass

    def handle_read(self):
        data = self.recv(8192)
        print data

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer[:])
        self.buffer = self.buffer[sent:]
CHAPTER
TWELVE

Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the internet. Some, like SGML and XML, may be useful for other applications as well.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>formatter</td>
<td>Generic output formatter and device interface.</td>
</tr>
<tr>
<td>rfc822</td>
<td>Parse RFC 822 style mail headers.</td>
</tr>
<tr>
<td>mime tools</td>
<td>Tools for parsing MIME-style message bodies.</td>
</tr>
<tr>
<td>MimeWriter</td>
<td>Generic MIME file writer.</td>
</tr>
<tr>
<td>mult file</td>
<td>Support for reading files which contain distinct parts, such as some MIME data.</td>
</tr>
<tr>
<td>binhex</td>
<td>Encode and decode files in binhex4 format.</td>
</tr>
<tr>
<td>uu</td>
<td>Encode and decode files in uuencode format.</td>
</tr>
</tbody>
</table>
| bin ascii     | Tools for converting between binary and various ASCII-encoded binary represen-
|              | tations.                                                                   |
| xdr library  | Encoders and decoders for the External Data Representation (XDR).           |
| mail cap     | Mailcap file handling.                                                      |
| mimetypes    | Mapping of filename extensions to MIME types.                              |
| base64       | Encode and decode files using the MIME base64 data.                        |
| quopri       | Encode and decode files using the MIME quoted-printable encoding.           |
| mailbox      | Read various mailbox formats.                                               |
| mh lib       | Manipulate MH mailboxes from Python.                                       |
| mimify       | Mimification and unmimification of mail messages.                          |
| netrc         | Loading of `.netrc` files.                                                  |
| robot parser | Accepts as input a list of lines or URL that refers to a robots.txt file,  |
|              | parses the file, then builds a set of rules from the file.                 |

12.1 formatter — Generic output formatting

This module supports two interface definitions, each with multiple implementations. The formatter interface is used by the HTMLParser class of the html library module, and the writer interface is required by the formatter interface.

Formatters transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of “change back” operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.
12.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

AS_IS

Value which can be used in the font specification passed to the push_font() method described below, or as the new value to any other push_property() method. Pushing the AS_IS value allows the corresponding pop_property() method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

**writer**

The writer instance with which the formatter interacts.

**end_paragraph(blanklines)**

Close any open paragraphs and insert at least blanklines before the next paragraph.

**add_line_break()**

Add a hard line break if one does not already exist. This does not break the logical paragraph.

**add_hor_rule(*args, **kw)**

Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer’s send_line_break() method.

**add_flowing_data(data)**

Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to add_flowing_data() is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

**add_literal_data(data)**

Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of data.

**add_label_data(format, counter)**

Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the format value is a string, it is interpreted as a format specification for counter, which should be an integer. The result of this formatting becomes the value of the label; if format is not a string it is used as the label value directly. The label value is passed as the only argument to the writer’s send_label_data() method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character ‘1’ represents the counter value formatter as an Arabic number, the characters ‘A’ and ‘a’ represent alphabetic representations of the counter value in upper and lower case, respectively, and ‘I’ and ‘i’ represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

**flush_softspace()**

Send any pending whitespace buffered from a previous call to add_flowing_data() to the associated writer object. This should be called before any direct manipulation of the writer object.

**push_alignment(alignment)**

Push a new alignment setting onto the alignment stack. This may be AS_IS if no change is desired. If the alignment value is changed from the previous setting, the writer’s new_alignment() method is called with the alignment value.
pop_alignment()
    Restore the previous alignment.

push_font( (size, italic, bold, teletype) )
    Change some or all font properties of the writer object. Properties which are not set to AS_IS are set to the
    values passed in while others are maintained at their current settings. The writer’s new_font() method is
called with the fully resolved font specification.

pop_font()
    Restore the previous font.

push_margin(margin)
    Increase the number of left margin indentations by one, associating the logical tag margin with the new indenta-
tion. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than
AS_IS are not sufficient to change the margin.

pop_margin()
    Restore the previous margin.

push_style(*styles)
    Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple
    representing the entire stack, including AS_IS values, is passed to the writer’s new_styles() method.

pop_style([n = 1])
    Pop the last n style specifications passed to push_style(). A tuple representing the revised stack, including
AS_IS values, is passed to the writer’s new_styles() method.

set_spacing(spacing)
    Set the spacing style for the writer.

assert_line_data([flag = 1])
    Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when
the writer has been manipulated directly. The optional flag argument can be set to false if the writer manipulations
produced a hard line break at the end of the output.

12.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes
without modification or subclassing.

class NullFormatter([writer])
    A formatter which does nothing. If writer is omitted, a NullWriter instance is created. No methods of the
writer are called by NullFormatter instances. Implementations should inherit from this class if implement-
ing a writer interface but don’t need to inherit any implementation.

class AbstractFormatter(writer)
    The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be
used directly in most circumstances. It has been used to implement a full-featured world-wide web browser.

12.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below
are the required interfaces which all writers must support once initialized. Note that while most applications can use
the AbstractFormatter class as a formatter, the writer must typically be provided by the application.

flush()
    Flush any buffered output or device control events.
**new_alignment** *(align)*
Set the alignment style. The *align* value can be any object, but by convention is a string or *None*, where *None* indicates that the writer’s “preferred” alignment should be used. Conventional *align* values are ‘left’, ‘center’, ‘right’, and ‘justify’.

**new_font** *(font)*
Set the font style. The value of *font* will be *None*, indicating that the device’s default font should be used, or a tuple of the form *(size, italic, bold, teletype)*. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are boolean indicators specifying which of those font attributes should be used.

**new_margin** *(margin, level)*
Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer’s discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

**new_spacing** *(spacing)*
Set the spacing style to *spacing*.

**new_styles** *(styles)*
Set additional styles. The *styles* value is a tuple of arbitrary values; the value *AS_IS* should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

**send_line_break** *(*)
Break the current line.

**send_paragraph** *(blankline)*
Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to **send_line_break** (*) before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

**send_hor_rule** *(args, **kw)*
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via **send_line_break** (*) .

**send_flowing_data** *(data)*
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

**send_literal_data** *(data)*
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the **send_formatted_data** () interface.

**send_label_data** *(data)*
Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

### 12.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the **NullWriter** class.
class **NullWriter** ()
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class **AbstractWriter** ()
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class **DumbWriter** ([file[, maxcol = 72 ]])
Simple writer class which writes output on the file object passed in as file or, if file is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by maxcol. This class is suitable for reflowing a sequence of paragraphs.

### 12.2  rfc822 — Parse RFC 822 mail headers

This module defines a class, **Message**, which represents a collection of “email headers” as defined by the Internet standard RFC 822. It is used in various contexts, usually to read such headers from a file. This module also defines a helper class **AddressList** for parsing RFC 822 addresses. Please refer to the RFC for information on the specific syntax of RFC 822 headers.

The **mailbox** module provides classes to read mailboxes produced by various end-user mail programs.

class **Message** (file[, seekable ])
A **Message** instance is instantiated with an input object as parameter. **Message** relies only on the input object having a **readline()** method; in particular, ordinary file objects qualify. Instantiation reads headers from the input object up to a delimiter line (normally a blank line) and stores them in the instance. The message body, following the headers, is not consumed.

This class can work with any input object that supports a **readline()** method. If the input object has seek and tell capability, the **rewindbody()** method will work; also, illegal lines will be pushed back onto the input stream. If the input object lacks seek but has an **unread()** method that can push back a line of input, **Message** will use that to push back illegal lines. Thus this class can be used to parse messages coming from a buffered stream.

The optional **seekable** argument is provided as a workaround for certain stdio libraries in which **tell()** discards buffered data before discovering that the **lseek()** system call doesn’t work. For maximum portability, you should set the seekable argument to zero to prevent that initial **tell()** when passing in an unseekable object such as a file object created from a socket object.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. **m['From'], m['from']** and **m['FROM']** all yield the same result.

class **AddressList** (field)
You may instantiate the **AddressList** helper class using a single string parameter, a comma-separated list of RFC 822 addresses to be parsed. (The parameter **None** yields an empty list.)

def **parsedate**(date)
Attempts to parse a date according to the rules in RFC 822. However, some mailers don’t follow that format as specified, so **parsedate()** tries to guess correctly in such cases. **date** is a string containing an RFC 822 date, such as ‘Mon, 20 Nov 1995 19:12:08 -0500’. If it succeeds in parsing the date, **parsedate()** returns a 9-tuple that can be passed directly to **time.mktime()**; otherwise **None** will be returned. Note that fields 6, 7, and 8 of the result tuple are not usable.

**parsedate_tz**(date)
Performs the same function as **parsedate()**, but returns either **None** or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to **time.mktime()**, and the tenth is the offset of the date’s timezone.

---

12.2. rfc822 — Parse RFC 822 mail headers
from UTC (which is the official term for Greenwich Mean Time). (Note that the sign of the timezone offset is the opposite of the sign of the \texttt{time.timezone} variable for the same timezone; the latter variable follows the POSIX standard while this module follows RFC 822.) If the input string has no timezone, the last element of the tuple returned is \texttt{None}. Note that fields 6, 7, and 8 of the result tuple are not usable.

\texttt{mktime\_tz (tuple)}

Turn a 10-tuple as returned by \texttt{parsedate\_tz()} into a UTC timestamp. It the timezone item in the tuple is \texttt{None}, assume local time. Minor deficiency: this first interprets the first 8 elements as a local time and then compensates for the timezone difference; this may yield a slight error around daylight savings time switch dates. Not enough to worry about for common use.

See Also:

Module \texttt{mailbox} (section 12.14):
Classes to read various mailbox formats produced by end-user mail programs.

Module \texttt{mimetypes} (section 12.3):
Subclass of \texttt{rfc.Message} that handles MIME encoded messages.

12.2.1 Message Objects

A \texttt{Message} instance has the following methods:

\texttt{rewindbody()} 
Seek to the start of the message body. This only works if the file object is seekable.

\texttt{isheader (line)}
Returns a line’s canonicalized fieldname (the dictionary key that will be used to index it) if the line is a legal RFC 822 header; otherwise returns \texttt{None} (implying that parsing should stop here and the line be pushed back on the input stream). It is sometimes useful to override this method in a subclass.

\texttt{islast (line)}
Return true if the given line is a delimiter on which \texttt{Message} should stop. The delimiter line is consumed, and the file object’s read location positioned immediately after it. By default this method just checks that the line is blank, but you can override it in a subclass.

\texttt{iscomment (line)}
Return true if the given line should be ignored entirely, just skipped. By default this is a stub that always returns false, but you can override it in a subclass.

\texttt{getallmatchingheaders (name)}
Return a list of lines consisting of all headers matching \texttt{name}, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches \texttt{name}.

\texttt{getfirstmatchingheader (name)}
Return a list of lines comprising the first header matching \texttt{name}, and its continuation line(s), if any. Return \texttt{None} if there is no header matching \texttt{name}.

\texttt{getrawheader (name)}
Return a single string consisting of the text after the colon in the first header matching \texttt{name}. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return \texttt{None} if there is no header matching \texttt{name}.

\texttt{getheader (name[, default ])}
Like \texttt{getrawheader (name)}, but strip leading and trailing whitespace. Internal whitespace is not stripped. The optional \texttt{default} argument can be used to specify a different default to be returned when there is no header matching \texttt{name}.

\texttt{get (name[, default ])}
An alias for \texttt{getheader()}, to make the interface more compatible with regular dictionaries.
getaddr(name)

Return a pair (full name, email address) parsed from the string returned by getheader(name). If no header matching name exists, return (None, None); otherwise both the full name and the address are (possibly empty) strings.

Example: If m's first From header contains the string 'Jack Jansen <jack@cwi.nl>', then m.getaddr('From') will yield the pair ('Jack Jansen', 'jack@cwi.nl'). If the header contained 'Jack Jansen <jack@cwi.nl>' instead, it would yield the exact same result.

getaddrlist(name)

This is similar to getaddr(list), but parses a header containing a list of email addresses (e.g. a To header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching name, return an empty list.

If multiple headers exist that match the named header (e.g. if there are several Cc headers), all are parsed for addresses. Any continuation lines the named headers contain are also parsed.

getdate(name)

Retrieve a header using getheader() and parse it into a 9-tuple compatible with time.mktime(); note that fields 6, 7, and 8 are not usable. If there is no header matching name, or it is unparsable, return None.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

getdate_tz(name)

Retrieve a header using getheader() and parse it into a 10-tuple; the first 9 elements will make a tuple compatible with time.mktime(), and the 10th is a number giving the offset of the date's timezone from UTC. Note that fields 6, 7, and 8 are not usable. Similarly to getdate(), if there is no header matching name, or it is unparsable, return None.

Message instances also support a read-only mapping interface. In particular: m[name] is like m.getheader(name) but raises KeyError if there is no matching header; and len(m), m.has_key(name), m.keys(), m.values() and m.items() act as expected (and consistently).

Finally, Message instances have two public instance variables:

headers
A list containing the entire set of header lines, in the order in which they were read (except that setitem calls may disturb this order). Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

fp
The file or file-like object passed at instantiation time. This can be used to read the message content.

12.2.2 AddressList Objects

An AddressList instance has the following methods:

__len__()  
Return the number of addresses in the address list.

__str__()  
Return a canonicalized string representation of the address list. Addresses are rendered in 'name'¡host@domain¿ form, comma-separated.

__add__(alist)

Return a new AddressList instance that contains all addresses in both AddressList operands, with duplicates removed (set union).

__iadd__(alist)

12.2. rfc822 — Parse RFC 822 mail headers
In-place version of \texttt{add}(); turns this \texttt{AddressList} instance into the union of itself and the right-hand instance, \texttt{alist}.

\texttt{sub}(\texttt{alist})

Return a new \texttt{AddressList} instance that contains every address in the left-hand \texttt{AddressList} operand that is not present in the right-hand address operand (set difference).

\texttt{isub}(\texttt{alist})

In-place version of \texttt{sub}(), removing addresses in this list which are also in \texttt{alist}.

Finally, \texttt{AddressList} instances have one public instance variable:

\texttt{addresslist}

A list of tuple string pairs, one per address. In each member, the first is the canonicalized name part, the second is the actual route-address ('@'-separated username-host.domain pair).

12.3 \texttt{mimetypes} — Tools for parsing MIME messages

This module defines a subclass of the \texttt{rfc822} module’s \texttt{Message} class and a number of utility functions that are useful for the manipulation for MIME multipart or encoded message.

It defines the following items:

\texttt{Message(\texttt{fp}, \texttt{seekable})}

Return a new instance of the \texttt{Message} class. This is a subclass of the \texttt{rfc822.Message} class, with some additional methods (see below). The \texttt{seekable} argument has the same meaning as for \texttt{rfc822.Message}.

\texttt{choose\_boundary()}

Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form ‘\texttt{hostipaddr.\texttt{uid}.\texttt{pid}.\texttt{timestamp}.\texttt{random}’.’

\texttt{decode(\texttt{input}, \texttt{output}, \texttt{encoding})}

Read data encoded using the allowed MIME \texttt{encoding} from open file object \texttt{input} and write the decoded data to open file object \texttt{output}. Valid values for \texttt{encoding} include ‘\texttt{base64}’, ‘\texttt{quoted-printable}’ and ‘\texttt{uuencode}’.

\texttt{encode(\texttt{input}, \texttt{output}, \texttt{encoding})}

Read data from open file object \texttt{input} and write it encoded using the allowed MIME \texttt{encoding} to open file object \texttt{output}. Valid values for \texttt{encoding} are the same as for \texttt{decode()}.

\texttt{copy\_literal(\texttt{input}, \texttt{output})}

Read lines from open file \texttt{input} until EOF and write them to open file \texttt{output}.

\texttt{copy\_binary(\texttt{input}, \texttt{output})}

Read blocks until EOF from open file \texttt{input} and write them to open file \texttt{output}. The block size is currently fixed at 8192.

See Also:

Module \texttt{rfc822} (section 12.2):

Provides the base class for \texttt{mimetypes.Message}.

Module \texttt{multifile} (section 12.5):

Support for reading files which contain distinct parts, such as MIME data.


The MIME Frequently Asked Questions document. For an overview of MIME, see the answer to question 1.1 in Part 1 of this document.
12.3.1 Additional Methods of Message Objects

The Message class defines the following methods in addition to the rfc822.Message methods:

**getplist()**
Return the parameter list of the content-type header. This is a list of strings. For parameters of the form ‘key=value’, key is converted to lower case but value is not. For example, if the message contains the header ‘Content-type: text/html; spam=1; Spam=2; Spam’ then getplist() will return the Python list ['spam=1', 'spam=2', 'Spam'].

**getparam(name)**
Return the value of the first parameter (as returned by getplist() of the form ‘name=value’ for the given name. If value is surrounded by quotes of the form ‘<...>’ or ‘"...”’, these are removed.

**getencoding()**
Return the encoding specified in the content-transfer-encoding message header. If no such header exists, return ‘7bit’. The encoding is converted to lower case.

**gettype()**
Return the message type (of the form ‘type/subtype’) as specified in the content-type header. If no such header exists, return ‘text/plain’. The type is converted to lower case.

**getmaintype()**
Return the main type as specified in the content-type header. If no such header exists, return ‘text’. The main type is converted to lower case.

**getsubtype()**
Return the subtype as specified in the content-type header. If no such header exists, return ‘plain’. The subtype is converted to lower case.

12.4 MimeWriter — Generic MIME file writer

This module defines the class MimeWriter. The MimeWriter class implements a basic formatter for creating MIME multi-part files. It doesn’t seek around the output file nor does it use large amounts of buffer space. You must write the parts out in the order that they should occur in the final file. MimeWriter does buffer the headers you add, allowing you to rearrange their order.

**class MimeWriter (fp)**
Return a new instance of the MimeWriter class. The only argument passed, fp, is a file object to be used for writing. Note that a StringIO object could also be used.

12.4.1 MimeWriter Objects

MimeWriter instances have the following methods:

**addheader (key, value[, prefix])**
Add a header line to the MIME message. The key is the name of the header, where the value obviously provides the value of the header. The optional argument prefix determines where the header is inserted; ‘1’ is insert at the start. The default is to append.

**flushheaders ()**
Causes all headers accumulated so far to be written out (and forgotten). This is useful if you don’t need a body part at all, e.g. for a subpart of type message/rfc822 that’s (mis)used to store some header-like information.

**startbody (ctype[, plist[, prefix]])**
Returns a file-like object which can be used to write to the body of the message. The content-type is set to the
provided *ctype*, and the optional parameter *plist* provides additional parameters for the content-type declaration. *prefix* functions as in `addheader()` except that the default is to insert at the start.

**startmultipartbody**(subtype[, boundary[, plist[, prefix]]])

Returns a file-like object which can be used to write to the body of the message. Additionally, this method initializes the multi-part code, where *subtype* provides the multipart subtype, *boundary* may provide a user-defined boundary specification, and *plist* provides optional parameters for the subtype. *prefix* functions as in `startbody()`.

**nextpart()**

Returns a new instance of MimeWriter which represents an individual part in a multipart message. This may be used to write the part as well as used for creating recursively complex multipart messages. The message must first be initialized with `startmultipartbody()` before using `nextpart()`.

**lastpart()**

This is used to designate the last part of a multipart message, and should *always* be used when writing multipart messages.

### 12.5 multiframe — Support for files containing distinct parts

The MultiFile object enables you to treat sections of a text file as file-like input objects, with ‘’ being returned by `readline()` when a given delimiter pattern is encountered. The defaults of this class are designed to make it useful for parsing MIME multipart messages, but by subclassing it and overriding methods it can be easily adapted for more general use.

**class MultiFile**(fp[, seekable ])

Create a multi-file. You must instantiate this class with an input object argument for the MultiFile instance to get lines from, such as as a file object returned by `open()`.

MultiFile only ever looks at the input object’s `readline()`, `seek()` and `tell()` methods, and the latter two are only needed if you want random access to the individual MIME parts. To use MultiFile on a non-seekable stream object, set the optional `seekable` argument to false; this will prevent using the input object’s `seek()` and `tell()` methods.

It will be useful to know that in MultiFile’s view of the world, text is composed of three kinds of lines: data, section-dividers, and end-markers. MultiFile is designed to support parsing of messages that may have multiple nested message parts, each with its own pattern for section-divider and end-marker lines.

#### 12.5.1 MultiFile Objects

A MultiFile instance has the following methods:

**readline**(str)

Read a line. If the line is data (not a section-divider or end-marker or real EOF) return it. If the line matches the most-recently-stacked boundary, return ‘’ and set `self.last` to 1 or 0 according as the match is or is not an end-marker. If the line matches any other stacked boundary, raise an error. On encountering end-of-file on the underlying stream object, the method raises Error unless all boundaries have been popped.

**readlines**(str)

Return all lines remaining in this part as a list of strings.

**read()**

Read all lines, up to the next section. Return them as a single (multiline) string. Note that this doesn’t take a size argument!

**seek**(pos[, whence ])

Seek. Seek indices are relative to the start of the current section. The *pos* and *whence* arguments are interpreted
as for a file seek.

**tell()**

Return the file position relative to the start of the current section.

**next()**

Skip lines to the next section (that is, read lines until a section-divider or end-marker has been consumed). Return true if there is such a section, false if an end-marker is seen. Re-enable the most-recently-pushed boundary.

**is_data(str)**

Return true if *str* is data and false if it might be a section boundary. As written, it tests for a prefix other than '---' at start of line (which all MIME boundaries have) but it is declared so it can be overridden in derived classes.

Note that this test is used intended as a fast guard for the real boundary tests; if it always returns false it will merely slow processing, not cause it to fail.

**push(str)**

Push a boundary string. When an appropriately decorated version of this boundary is found as an input line, it will be interpreted as a section-divider or end-marker. All subsequent reads will return the empty string to indicate end-of-file, until a call to **pop()** removes the boundary a or **next()** call reenables it.

It is possible to push more than one boundary. Encountering the most-recently-pushed boundary will return EOF; encountering any other boundary will raise an error.

**pop()**

Pop a section boundary. This boundary will no longer be interpreted as EOF.

**section_divider(str)**

Turn a boundary into a section-divider line. By default, this method prepends '---' (which MIME section boundaries have) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF; as comparison with the result ignores trailing whitespace.

**end_marker(str)**

Turn a boundary string into an end-marker line. By default, this method prepends '---' and appends '---' (like a MIME-multipart end-of-message marker) but it is declared so it can be be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

Finally, **MultiFile** instances have two public instance variables:

**level**

Nesting depth of the current part.

**last**

True if the last end-of-file was for an end-of-message marker.
12.5.2 MultiFile Example

import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
    """Return the first element in a multipart MIME message on stream
    matching mimetype.""

    msg = mimetools.Message(stream)
   (msgtype = msg.gettype())
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":

        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()

12.6 binhex — Encode and decode binhex4 files

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. On the Macintosh, both forks of a file and the finder information are encoded (or decoded), on other platforms only the data fork is handled.

The binhex module defines the following functions:

binhex (input, output)
    Convert a binary file with filename input to binhex file output. The output parameter can either be a filename or a file-like object (any object supporting a write() and close() method).

hexbin (input[, output])
    Decode a binhex file input. input may be a filename or a file-like object supporting read() and close() methods. The resulting file is written to a file named output, unless the argument is omitted in which case the output filename is read from the binhex file.

See Also:
Module binascii (section 12.8):
    Support module containing ASCII-to-binary and binary-to-ASCII conversions.
12.6.1 Notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the Macintosh newline convention (carriage-return as end of line).

As of this writing, \texttt{hexbin()} appears to not work in all cases.

12.7 \texttt{uu} — Encode and decode uuencode files

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ascii-only connections. Wherever a file argument is expected, the methods accept a file-like object. For backwards compatibility, a string containing a pathname is also accepted, and the corresponding file will be opened for reading and writing; the pathname \texttt{'}-\texttt{'} is understood to mean the standard input or output. However, this interface is deprecated; it's better for the caller to open the file itself, and be sure that, when required, the mode is \texttt{'rb'} or \texttt{'wb'} on Windows or DOS.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The \texttt{uu} module defines the following functions:

\begin{verbatim}
encode (in_file, out_file[, name[, mode]])
    Uuencode file \texttt{in_file} into file \texttt{out_file}. The uuencoded file will have the header specifying \texttt{name} and \texttt{mode} as the defaults for the results of decoding the file. The default defaults are taken from \texttt{in_file}, or \texttt{'}-\texttt{'} and \texttt{0666} respectively.

decode (in_file[, out_file[, mode]])
    This call decodes uuencoded file \texttt{in_file} placing the result on file \texttt{out_file}. If \texttt{out_file} is a pathname, \texttt{mode} is used to set the permission bits if the file must be created. Defaults for \texttt{out_file} and \texttt{mode} are taken from the uuencode header.
\end{verbatim}

See Also:

\texttt{Module binascii (section 12.8)}:
Support module containing ASCII-to-binary and binary-to-ASCII conversions.

12.8 \texttt{binascii} — Convert between binary and ASCII

The \texttt{binascii} module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like \texttt{uu} or \texttt{binhex} instead, this module solely exists because bit-manipulation of large amounts of data is slow in Python.

The \texttt{binascii} module defines the following functions:

\begin{verbatim}
a2b_uu (string)
    Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

b2a_uu (data)
    Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of \texttt{data} should be at most 45.

a2b_base64 (string)
    Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

b2a_base64 (data)
\end{verbatim}
Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The length of *data* should be at most 57 to adhere to the base64 standard.

**a2b_hqx** (*string*)
Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

**rledecode_hqx** (*data*)
Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses 0x90 after a byte as a repeat indicator, followed by a count. A count of 0 specifies a byte value of 0x90. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the Incomplete exception is raised.

**rlecode_hqx** (*data*)
Perform binhex4 style RLE-compression on *data* and return the result.

**b2a_hqx** (*data*)
Perform hexbin4 binary-to-ASCII translation and return the resulting string. The argument should already be RLE-coded, and have a length divisible by 3 (except possibly the last fragment).

**crc_hqx** (*data, crc*)
Compute the binhex4 crc value of *data*, starting with an initial *crc* and returning the result.

**crc32** (*data*, *crc*)
Compute CRC-32, the 32-bit checksum of *data*, starting with an initial *crc*. This is consistent with the ZIP file checksum. Use as follows:

```python
print binascii.crc32("hello world")
# Or, in two pieces:
crc = binascii.crc32("hello")
crc = binascii.crc32(" world", crc)
print crc
```

**b2a_hex** (*data*)
**hexlify** (*data*)
Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The resulting string is therefore twice as long as the length of *data*.

**a2b_hex** (*hexstr*)
**unhexlify** (*hexstr*)
Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of *b2a_hex()* (*hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise a TypeError is raised.

**exception Error**
Exception raised on errors. These are usually programming errors.

**exception Incomplete**
Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

See Also:
* Module base64 (section 12.12):
  Support for base64 encoding used in MIME email messages.
* Module binhex (section 12.6):
  Support for the binhex format used on the Macintosh.
* Module uu (section 12.7):
Support for UU encoding used on UNIX.

12.9  xdrlib — Encode and decode XDR data

The xdrlib module supports the External Data Representation Standard as described in RFC 1014, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The xdrlib module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

class Packer()

Packer is the class for packing data into XDR representation. The Packer class is instantiated with no arguments.

class Unpacker(data)

Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as data.

See Also:

RFC 1014, “XDR: External Data Representation Standard’

This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by RFC 1832.

RFC 1832, “XDR: External Data Representation Standard’

Newer RFC that provides a revised definition of XDR.

12.9.1  Packer Objects

Packer instances have the following methods:

get_buffer()

Returns the current pack buffer as a string.

reset()

Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate pack_type() method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: pack_uint(), pack_int(), pack_enum(), pack_bool(), pack_uhyper(), and pack_hyper().

pack_float(value)

Packs the single-precision floating point number value.

pack_double(value)

Packs the double-precision floating point number value.

The following methods support packing strings, bytes, and opaque data:

pack_fstring(n, s)

Packs a fixed length string, s. n is the length of the string but it is not packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

pack_fopaque(n, data)

Packs a fixed length opaque data stream, similarly to pack_fstring().

pack_string(s)

Packs a variable length string, s. The length of the string is first packed as an unsigned integer, then the string
data is packed with pack_fstring().

**pack_opaque** *(data)*

Packs a variable length opaque data string, similarly to pack_string().

**pack_bytes** *(bytes)*

Packs a variable length byte stream, similarly to pack_string().

The following methods support packing arrays and lists:

**pack_list** *(list, pack_item)*

Packs a *list* of homogeneous items. This method is useful for lists with an indeterminate size; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer 1 is packed first, followed by the data value from the list. *pack_item* is the function that is called to pack the individual item. At the end of the list, an unsigned integer 0 is packed.

For example, to pack a list of integers, the code might appear like this:

```python
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

**pack_farray** *(n, array, pack_item)*

Packs a fixed length list *(array)* of homogeneous items. *n* is the length of the list; it is *not* packed into the buffer, but a ValueError exception is raised if len*(array)* is not equal to *n*. As above, *pack_item* is the function used to pack each element.

**pack_array** *(list, pack_item)*

Packs a variable length *list* of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in pack_farray() above.

### 12.9.2 Unpacker Objects

The *Unpacker* class offers the following methods:

**reset** *(data)*

Resets the string buffer with the given *data*.

**get_position** ()

Returns the current unpack position in the data buffer.

**set_position** *(position)*

Sets the data buffer unpack position to *position*. You should be careful about using get_position() and set_position().

**get_buffer** ()

Returns the current unpack data buffer as a string.

**done** ()

Indicates unpack completion. Raises an Error exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a *Packer*, can be unpacked with an *Unpacker*. Unpacking methods are of the form unpack_type(), and take no arguments. They return the unpacked object.

**unpack_float** ()

Unpacks a single-precision floating point number.

**unpack_double** ()

Unpacks a double-precision floating point number, similarly to unpack_float().

In addition, the following methods unpack strings, bytes, and opaque data:
unpack_fstring(n)
Unpacks and returns a fixed length string. n is the number of characters expected. Padding with null bytes to
guaranteed 4 byte alignment is assumed.

unpack_fopaque(n)
Unpacks and returns a fixed length opaque data stream, similarly to unpack_fstring().

unpack_string()
Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer,
then the string data is unpacked with unpack_fstring().

unpack_opaque()
Unpacks and returns a variable length opaque data string, similarly to unpack_string().

unpack_bytes()
Unpacks and returns a variable length byte stream, similarly to unpack_string().

The following methods support unpacking arrays and lists:

unpack_list(unpack_item)
Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking
an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates
the end of the list. unpack_item is the function that is called to unpack the items.

unpack_farray(n, unpack_item)
Unpacks and returns (as a list) a fixed length array of homogeneous items. n is number of list elements to expect
in the buffer. As above, unpack_item is the function used to unpack each element.

unpack_array(unpack_item)
Unpacks and returns a variable length list of homogeneous items. First, the length of the list is unpacked as an
unsigned integer, then each element is unpacked as in unpack_farray() above.

12.9.3 Exceptions

Exceptions in this module are coded as class instances:

exception Error
The base exception class. Error has a single public data member msg containing the description of the error.

exception ConversionError
Class derived from Error. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:

```python
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError, instance:
    print 'packing the double failed:', instance.msg
```

12.10 mailcap — Mailcap file handling.

Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to
files with different MIME types. (The name “mailcap” is derived from the phrase “mail capability”.) For example, a
mailcap file might contain a line like 'video/mpeg; xmpeg %s'. Then, if the user encounters an email message

12.10. mailcap — Mailcap file handling. 343
or Web document with the MIME type `video/mpeg`, `%s` will be replaced by a filename (usually one belonging to a temporary file) and the `xmpeg` program can be automatically started to view the file.

The mailcap format is documented in RFC 1524, “A User Agent Configuration Mechanism For Multimedia Mail Format Information,” but is not an Internet standard. However, mailcap files are supported on most UNIX systems.

```python
def findmatch(caps, MIMEtype, key, filename, plist):
    return a 2-tuple; the first element is a string containing the command line to be executed (which can be passed to `os.system()`), and the second element is the mailcap entry for a given MIME type. If no matching MIME type can be found, (None, None) is returned.

    key is the name of the field desired, which represents the type of activity to be performed; the default value is 'view', since in the most common case you simply want to view the body of the MIME-typed data. Other possible values might be 'compose' and 'edit', if you wanted to create a new body of the given MIME type or alter the existing body data. See RFC 1524 for a complete list of these fields.

    filename is the filename to be substituted for '%s' in the command line; the default value is '/dev/null' which is almost certainly not what you want, so usually you'll override it by specifying a filename.

    plist can be a list containing named parameters; the default value is simply an empty list. Each entry in the list must be a string containing the parameter name, an equals sign (=), and the parameter's value. Mailcap entries can contain named parameters like '%{foo}', which will be replaced by the value of the parameter named 'foo'. For example, if the command line 'showpartial %{id} %{number} %{total}' was in a mailcap file, and `plist` was set to '['id=1', 'number=2', 'total=3']`, the resulting command line would be "showpartial 1 2 3".

    In a mailcap file, the "test" field can optionally be specified to test some external condition (e.g., the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. `findmatch()` will automatically check such conditions and skip the entry if the check fails.

```

```python
def getcaps()
    Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the `findmatch()` function. An entry is stored as a list of dictionaries, but it shouldn't be necessary to know the details of this representation.

    The information is derived from all of the mailcap files found on the system. Settings in the user's mailcap file `$HOME/.mailcap` will override settings in the system mailcap files `/etc/mailcap`, `/usr/etc/mailcap`, and `/usr/local/etc/mailcap`.

    An example usage:

```python
>>> import mailcap
>>> d=mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='/tmp/tmp1223')
('xmpeg /tmp/tmp1223', {'view': 'xmpeg %s'})
```

12.11 mimetypes — Map filenames to MIME types

The `mimetypes` converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the later conversion.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call `init()`.

```python
def guess_type(filename)
    Guess the type of a file based on its filename or URL, given by `filename`. The return value is a tuple `(type,`
where type is None if the type can’t be guessed (no or unknown suffix) or a string of the form ‘type/subtype’, usable for a MIME content-type header; and encoding is None for no encoding or the name of the program used to encode (e.g. compress or gzip). The encoding is suitable for use as a content-encoding header, not as a content-transfer-encoding header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitive, then case insensitive.

guess_extension(type)
Guess the extension for a file based on its MIME type, given by type. The return value is a string giving a filename extension, including the leading dot (‘.’). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type type by guess_type(). If no extension can be guessed for type, None is returned.

Some additional functions and data items are available for controlling the behavior of the module.

init([files])
Initialize the internal data structures. If given, files must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from knownfiles. Each file named in files or knownfiles takes precedence over those named before it. Calling init() repeatedly is allowed.

read_mime_types(filename)
Load the type map given in the file filename, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (‘.’), to strings of the form ‘type/subtype’. If the file filename does not exist or cannot be read, None is returned.

inited
Flag indicating whether or not the global data structures have been initialized. This is set to true by init().

knownfiles
List of type map file names commonly installed. These files are typically named ‘mime.types’ and are installed in different locations by different packages.

suffix_map
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the ‘.tgz’ extension is mapped to ‘.tar.gz’ to allow the encoding and type to be recognized separately.

encodings_map
Dictionary mapping filename extensions to encoding types.

types_map
Dictionary mapping filename extensions to MIME types.

## 12.12 base64 — Encode and decode MIME base64 data

This module performs base64 encoding and decoding of arbitrary binary strings into text strings that can be safely emailed or posted. The encoding scheme is defined in RFC 1521 (MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies, section 5.2, “Base64 Content-Transfer-Encoding”) and is used for MIME email and various other Internet-related applications; it is not the same as the output produced by the uuencode program. For example, the string ‘www.python.org’ is encoded as the string ‘d3d3LnB5dGhvbi5vcmc=
’. 

decode(input, output)
Decode the contents of the input file and write the resulting binary data to the output file. input and output must either be file objects or objects that mimic the file object interface. input will be read until input.read() returns an empty string.

decodestring(s)
Decode the string s, which must contain one or more lines of base64 encoded data, and return a string containing
the resulting binary data.

**encode** *(input, output)*

Encode the contents of the *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string.

**encode_string**(s)

Encode the string *s*, which can contain arbitrary binary data, and return a string containing one or more lines of base64 encoded data.

See Also:

Module binascii (section 12.8):

Support module containing ASCII-to-binary and binary-to-ASCII conversions.

RFC 1521, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”.

Section 5.2, “Base64 Content-Transfer-Encoding,” provides the definition of the base64 encoding.

12.13  **quopri** — Encode and decode MIME quoted-printable data

This module performs quoted-printable transport encoding and decoding, as defined in RFC 1521: “MIME (Multipurpose Internet Mail Extensions) Part One”. The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the base64 module is more compact if there are many such characters, as when sending a graphics file.

**decode** *(input, output)*

Decode the contents of the *input* file and write the resulting decoded binary data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string.

**encode** *(input, output, quotetabs)*

Encode the contents of the *input* file and write the resulting quoted-printable data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until *input.read()* returns an empty string.

See Also:

Module mimify (section 12.16):

General utilities for processing of MIME messages.

12.14  **mailbox** — Read various mailbox formats

This module defines a number of classes that allow easy and uniform access to mail messages in a (UNIX) mailbox.

**class**  **UnixMailbox**(fp[, factory])

Access to a classic UNIX-style mailbox, where all messages are contained in a single file and separated by ‘From ’ (a.k.a. ‘From’) lines. The file object *fp* points to the mailbox file. The optional *factory* parameter is a callable that should create new message objects. *factory* is called with one argument, *fp* by the *next()* method of the mailbox object. The default is the rfc822.Message class (see the rfc822 module).

For maximum portability, messages in a UNIX-style mailbox are separated by any line that begins exactly with the string ‘From ’ (note the trailing space) if preceded by exactly two newlines. Because of the wide-range of variations in practice, nothing else on the From_ line should be considered. However, the current implementation doesn’t check for the leading two newlines. This is usually fine for most applications.

The UnixMailbox class implements a more strict version of From_ line checking, using a regular expression that usually correctly matched From_ delimiters. It considers delimiter line to be separated by ‘From name
time’ lines. For maximum portability, use the PortableUnixMailbox class instead. This class is identical to UnixMailbox except that individual messages are separated by only ‘From ’ lines.

For more information, see Configuring Netscape Mail on UNIX: Why the Content-Length Format is Bad.

class PortableUnixMailbox (fp[, factory ])
A less-strict version of UnixMailbox, which considers only the ‘From ’ at the beginning of the line separating messages. The “name time” portion of the From line is ignored, to protect against some variations that are observed in practice. This works since lines in the message which begin with ‘From ’ are quoted by mail handling software well before delivery.

class MmdfMailbox (fp[, factory ])
Access an MMDF-style mailbox, where all messages are contained in a single file and separated by lines consisting of 4 control-A characters. The file object fp points to the mailbox file. Optional factory is as with the UnixMailbox class.

class MHMailbox (dirname[, factory ])
Access an MH mailbox, a directory with each message in a separate file with a numeric name. The name of the mailbox directory is passed in dirname. factory is as with the UnixMailbox class.

class Maildir (dirname[, factory ])
Access a Qmail mail directory. All new and current mail for the mailbox specified by dirname is made available. factory is as with the UnixMailbox class.

class BabylMailbox (fp[, factory ])
Access a Babyl mailbox, which is similar to an MMDF mailbox. In Babyl format, each message has two sets of headers, the original headers and the visible headers. The original headers appear before a line containing only ‘*** EOOH ***’ (End-Of-Original-Headers) and the visible headers appear after the EOOH line. Babyl-compliant mail readers will show you only the visible headers, and BabylMailbox objects will return messages containing only the visible headers. You’ll have to do your own parsing of the mailbox file to get at the original headers. Mail messages start with the EOOH line and end with a line containing only ‘\037\014’.

factory is as with the UnixMailbox class.

12.14.1 Mailbox Objects

All implementations of Mailbox objects have one externally visible method:

next ()
Return the next message in the mailbox, created with the optional factory argument passed into the mailbox object’s constructor. By default this is an rfc822.Message object (see the rfc822 module). Depending on the mailbox implementation the fp attribute of this object may be a true file object or a class instance simulating a file object, taking care of things like message boundaries if multiple mail messages are contained in a single file, etc. If no more messages are available, this method returns None.

12.15 mhlib — Access to MH mailboxes

The mhlib module provides a Python interface to MH folders and their contents.

The module contains three basic classes, MH, which represents a particular collection of folders, Folder, which represents a single folder, and Message, which represents a single message.

class MH ([path[, profile ]])
MH represents a collection of MH folders.

class Folder (mh, name)
The Folder class represents a single folder and its messages.
class Message (folder, number[, name])

    Message objects represent individual messages in a folder. The Message class is derived from mimetools.Message.

12.15.1 MH Objects

MH instances have the following methods:

error (format[, ...])

    Print an error message – can be overridden.

getprofile (key)

    Return a profile entry (None if not set).

getpath ()

    Return the mailbox pathname.

gectext ()

    Return the current folder name.

setcontext (name)

    Set the current folder name.

listfolders ()

    Return a list of top-level folders.

listallfolders ()

    Return a list of all folders.

listsubfolders (name)

    Return a list of direct subfolders of the given folder.

listallsubfolders (name)

    Return a list of all subfolders of the given folder.

makefolder (name)

    Create a new folder.

deletefolder (name)

    Delete a folder – must have no subfolders.

openfolder (name)

    Return a new open folder object.

12.15.2 Folder Objects

Folder instances represent open folders and have the following methods:

error (format[, ...])

    Print an error message – can be overridden.

getfullname ()

    Return the folder’s full pathname.

getsequencesfilename ()

    Return the full pathname of the folder’s sequences file.

getmessagefilename (n)

    Return the full pathname of message n of the folder.

listmessages ()

348 Chapter 12. Internet Data Handling
Return a list of messages in the folder (as numbers).

**getcurrent()**
Return the current message number.

**setcurrent(n)**
Set the current message number to n.

**parsesequence(seq)**
Parse msgs syntax into list of messages.

**getlast()**
Get last message, or 0 if no messages are in the folder.

**setlast(n)**
Set last message (internal use only).

**getsequences()**
Return dictionary of sequences in folder. The sequence names are used as keys, and the values are the lists of message numbers in the sequences.

**putsequences(dict)**
Return dictionary of sequences in folder name: list.

**removemessages(list)**
Remove messages in list from folder.

**refilemessages(list, tofolder)**
Move messages in list to other folder.

**movemessage(n, tofolder, ton)**
Move one message to a given destination in another folder.

**copymessage(n, tofolder, ton)**
Copy one message to a given destination in another folder.

### 12.15.3 Message Objects

The Message class adds one method to those of `mimetools.Message`:

**openmessage(n)**
Return a new open message object (costs a file descriptor).

### 12.16 mimify — MIME processing of mail messages

The mimify module defines two functions to convert mail messages to and from MIME format. The mail message can be either a simple message or a so-called multipart message. Each part is treated separately. Mimifying (a part of) a message entails encoding the message as quoted-printable if it contains any characters that cannot be represented using 7-bit ASCII. Unmimifying (a part of) a message entails undoing the quoted-printable encoding. Mimify and unmimify are especially useful when a message has to be edited before being sent. Typical use would be:

```python
unmimify message
edit message
mimify message
send message
```

The modules defines the following user-callable functions and user-settable variables:
**mimify (infile, outfile)**

Copy the message in `infile` to `outfile`, converting parts to quoted-printable and adding MIME mail headers when necessary. `infile` and `outfile` can be file objects (actually, any object that has a `readline()` method (for `infile`) or a `write()` method (for `outfile`)) or strings naming the files. If `infile` and `outfile` are both strings, they may have the same value.

**unmimify (infile, outfile[, decode_base64])**

Copy the message in `infile` to `outfile`, decoding all quoted-printable parts. `infile` and `outfile` can be file objects (actually, any object that has a `readline()` method (for `infile`) or a `write()` method (for `outfile`)) or strings naming the files. If `infile` and `outfile` are both strings, they may have the same value. If the `decode_base64` argument is provided and tests true, any parts that are coded in the base64 encoding are decoded as well.

**mime_decode_header (line)**

Return a decoded version of the encoded header line in `line`.

**mime_encode_header (line)**

Return a MIME-encoded version of the header line in `line`.

**MAXLEN**

By default, a part will be encoded as quoted-printable when it contains any non-ASCII characters (i.e., characters with the 8th bit set), or if there are any lines longer than `MAXLEN` characters (default value 200).

**CHARSET**

When not specified in the mail headers, a character set must be filled in. The string used is stored in `CHARSET`, and the default value is ISO-8859-1 (also known as Latin1 (latin-one)).

This module can also be used from the command line. Usage is as follows:

```
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

to encode (mimify) and decode (unmimify) respectively. `infile` defaults to standard input, `outfile` defaults to standard output. The same file can be specified for input and output.

If the `-l` option is given when encoding, if there are any lines longer than the specified `length`, the containing part will be encoded.

If the `-b` option is given when decoding, any base64 parts will be decoded as well.

**See Also:**

Module `quopri` (section 12.13):
Encode and decode MIME quoted-printable files.

12.17 `netrc` — `netrc` file processing

New in version 1.5.2.

The `netrc` class parses and encapsulates the `netrc` file format used by the UNIX `ftp` program and other FTP clients.

**class netrc ([file])**

A `netrc` instance or subclass instance encapsulates data from a `netrc` file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `~/.netrc` in the user’s home directory will be read. Parse errors will raise `NetrcParseError` with diagnostic information including the file name, line number, and terminating token.

**exception NetrcParseError**

Exception raised by the `netrc` class when syntactical errors are encountered in source text. Instances of this
exception provide three interesting attributes: msg is a textual explanation of the error, filename is the name of the source file, and lineno gives the line number on which the error was found.

12.17.1 netrc Objects

A netrc instance has the following methods:

authenticators (host)
   Return a 3-tuple (login, account, password) of authenticators for host. If the netrc file did not contain an entry for the given host, return the tuple associated with the ‘default’ entry. If neither matching host nor default entry is available, return None.

__repr__()
   Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of netrc have public instance variables:

hosts
   Dictionary mapping host names to (login, account, password) tuples. The ‘default’ entry, if any, is represented as a pseudo-host by that name.

macros
   Dictionary mapping macro names to string lists.

12.18 robotparser — Parser for robots.txt

This module provides a single class, RobotFileParser, which answers questions about whether or not a particular user agent can fetch a URL on the web site that published the ‘robots.txt’ file. For more details on the structure of ‘robots.txt’ files, see http://info.webcrawler.com/mak/projects/robots/norobots.html.

class RobotFileParser()
   This class provides a set of methods to read, parse and answer questions about a single ‘robots.txt’ file.

   set_url (url)
      Sets the URL referring to a ‘robots.txt’ file.

   read()
      Reads the ‘robots.txt’ URL and feeds it to the parser.

   parse (lines)
      Parses the lines argument.

   can_fetch (useragent, url)
      Returns true if the useragent is allowed to fetch the url according to the rules contained in the parsed ‘robots.txt’ file.

   mtime()
      Returns the time the robots.txt file was last fetched. This is useful for long-running web spiders that need to check for new robots.txt files periodically.

   modified()
      Sets the time the robots.txt file was last fetched to the current time.

The following example demonstrates basic use of the RobotFileParser class.
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
0
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
1
Structured Markup Processing Tools

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

- **sgmllib**: Only as much of an SGML parser as needed to parse HTML.
- **htmllib**: A parser for HTML documents.
- **htmlentitydefs**: Definitions of HTML general entities.
- **xml.parsers.expat**: An interface to the Expat non-validating XML parser.
- **xml.dom**: Document Object Model API for Python.
- **xml.dom.minidom**: Lightweight Document Object Model (DOM) implementation.
- **xml.dom.pulldom**: Support for building partial DOM trees from SAX events.
- **xml.sax**: Package containing SAX2 base classes and convenience functions.
- **xml.sax.handler**: Base classes for SAX event handlers.
- **xml.sax.xmlreader**: Interface which SAX-compliant XML parsers must implement.
- **xmllib**: A parser for XML documents.

### 13.1 sgmllib — Simple SGML parser

This module defines a class `SGMLParser` which serves as the basis for parsing text files formatted in SGML (Standard Generalized Markup Language). In fact, it does not provide a full SGML parser — it only parses SGML insofar as it is used by HTML, and the module only exists as a base for the `htmllib` module.

```python
class SGMLParser:
    # The SGMLParser class is instantiated without arguments. The parser is hardcoded to recognize the following
    # constructs:
    #   • Opening and closing tags of the form `<tag attr="value" ...>` and `</tag>`, respectively.
    #   • Numeric character references of the form `& #name;`.
    #   • Entity references of the form `&name;`.
    #   • SGML comments of the form `<!--text-->`. Note that spaces, tabs, and newlines are allowed between
    #     the trailing `-->` and the immediately preceding `-->`.

    def reset(self):
        # Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

    def setnomoretags(self):
        # Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML
```

353
setliteral()
   Enter literal mode (CDA TA mode).

feed (data)
   Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is
   buffered until more data is fed or close() is called.

close()
   Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined
   by a derived class to define additional processing at the end of the input, but the redefined version should always
   call close().

get_starttag_text()
   Return the text of the most recently opened start tag. This should not normally be needed for structured process-
   ing, but may be useful in dealing with HTML “as deployed” or for re-generating input with minimal changes
   (whitespace between attributes can be preserved, etc.).

handle_starttag (tag, method, attributes)
   This method is called to handle start tags for which either a start_tag() or do_tag() method has been
   defined. The tag argument is the name of the tag converted to lower case, and the method argument is the
   bound method which should be used to support semantic interpretation of the start tag. The attributes argu-
   ment is a list of (name, value) pairs containing the attributes found inside the tag’s <> brackets. The
   name has been translated to lower case and double quotes and backslashes in the value have been inter-
   preted. For instance, for the tag <A HREF="http://www.cwi.nl/">, this method would be called as
   ‘unknown_starttag(’a’, [{’href’, ’http://www.cwi.nl/’}])’. The base implementation
   simply calls method with attributes as the only argument.

handle_endtag (tag, method)
   This method is called to handle endtags for which an end_tag() method has been defined. The tag argument
   is the name of the tag converted to lower case, and the method argument is the bound method which should
   be used to support semantic interpretation of the end tag. If no end_tag() method is defined for the closing
   element, this handler is not called. The base implementation simply calls method.

handle_data (data)
   This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class
   implementation does nothing.

handle_charref (ref)
   This method is called to process a character reference of the form ‘&#ref;’. In the base implementa-
   tion, ref must be a decimal number in the range 0-255. It translates the character to ASCII and calls the
   method handle_data() with the character as argument. If ref is invalid or out of range, the method
   unknown_charref (ref) is called to handle the error. A subclass must override this method to provide
   support for named character entities.

handle_entityref (ref)
   This method is called to process a general entity reference of the form ‘&ref;’ where ref is an general entity
   reference. It looks for ref in the instance (or class) variable entitydefs which should be a mapping from
   entity names to corresponding translations. If a translation is found, it calls the method handle_data() with
   the translation; otherwise, it calls the method unknown_entityref (ref). The default entitydefs
   defines translations for &amp;, &apos, &gt;, &lt;, and &quot;.

handle_comment (comment)
   This method is called when a comment is encountered. The comment argument is a string containing the
   text between the ‘<!--’ and ‘--’ delimiters, but not the delimiters themselves. For example, the comment
   ‘<!--text-->’ will cause this method to be called with the argument ‘text’. The default method does
   nothing.

handle_decl (data)
Method called when an SGML declaration is read by the parser. In practice, the DOCTYPE declaration is the only thing observed in HTML, but the parser does not discriminate among different (or broken) declarations. Internal subsets in a DOCTYPE declaration are not supported. The data parameter will be the entire contents of the declaration inside the <!...> markup. The default implementation does nothing.

**report_unbalanced**(tag)
This method is called when an end tag is found which does not correspond to any open element.

**unknown_starttag**(tag, attributes)
This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_endtag**(tag)
This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_charref**(ref)
This method is called to process unresolvable numeric character references. Refer to handle_charref() to determine what is handled by default. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_entityref**(ref)
This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the tag occurring in method names must be in lower case:

**start_tag**(attributes)
This method is called to process an opening tag tag. It has preference over do_tag(). The attributes argument has the same meaning as described for handle_starttag() above.

**do_tag**(attributes)
This method is called to process an opening tag tag that does not come with a matching closing tag. The attributes argument has the same meaning as described for handle_starttag() above.

**end_tag**()
This method is called to process a closing tag tag.

Note that the parser maintains a stack of open elements for which no end tag has been found yet. Only tags processed by start_tag() are pushed on this stack. Definition of an end_tag() method is optional for these tags. For tags processed by do_tag() or by unknown_tag(), no end_tag() method must be defined; if defined, it will not be used. If both start_tag() and do_tag() methods exist for a tag, the start_tag() method takes precedence.

13.2 htmllib — A parser for HTML documents

This module defines a class which can serve as a base for parsing text files formatted in the HyperText Mark-up Language (HTML). The class is not directly concerned with I/O — it must be provided with input in string form via a method, and makes calls to methods of a “formatter” object in order to produce output. The HTMLParser class is designed to be used as a base class for other classes in order to add functionality, and allows most of its methods to be extended or overridden. In turn, this class is derived from and extends the SGMLParser class defined in module sgmllib. The HTMLParser implementation supports the HTML 2.0 language as described in RFC 1866. Two implementations of formatter objects are provided in the formatter module; refer to the documentation for that module for information on the formatter interface.

The following is a summary of the interface defined by sgmllib.SGMLParser:
• The interface to feed data to an instance is through the `feed()` method, which takes a string argument. This can be called with as little or as much text at a time as desired; `'p.feed(a); p.feed(b)'` has the same effect as `'p.feed(a+b)'`. When the data contains complete HTML tags, these are processed immediately; incomplete elements are saved in a buffer. To force processing of all unprocessed data, call the `close()` method.

For example, to parse the entire contents of a file, use:

```python
parser.feed(open('myfile.html').read())
parser.close()
```

• The interface to define semantics for HTML tags is very simple: derive a class and define methods called `start_tag()`, `end_tag()`, or `do_tag()`. The parser will call these at appropriate moments: `start_tag()` or `do_tag()` is called when an opening tag of the form `<tag ...>` is encountered; `end_tag()` is called when a closing tag of the form `<tag>` is encountered. If an opening tag requires a corresponding closing tag, like `<H1>` ...

The module defines a single class:

```python
class HTMLParser(formatter)
```

This is the basic HTML parser class. It supports all entity names required by the HTML 2.0 specification (RFC 1866). It also defines handlers for all HTML 2.0 and many HTML 3.0 and 3.2 elements.

See Also:

- Module `htmlentitydefs` (section 13.3):
  Definition of replacement text for HTML 2.0 entities.
- Module `sgmllib` (section 13.1):
  Base class for `HTMLParser`.

### 13.2.1 HTMLParser Objects

In addition to tag methods, the `HTMLParser` class provides some additional methods and instance variables for use within tag methods.

- `formatter`
  This is the formatter instance associated with the parser.

- `nofill`
  Boolean flag which should be true when whitespace should not be collapsed, or false when it should be. In general, this should only be true when character data is to be treated as “preformatted” text, as within a `<PRE>` element. The default value is false. This affects the operation of `handle_data()` and `save_end()`.

- `anchor_bgn(href, name, type)`
  This method is called at the start of an anchor region. The arguments correspond to the attributes of the `<A>` tag with the same names. The default implementation maintains a list of hyperlinks (defined by the `HREF` attribute for `<A>` tags) within the document. The list of hyperlinks is available as the data attribute `anchorlist`.

- `anchor_end()`
  This method is called at the end of an anchor region. The default implementation adds a textual footnote marker using an index into the list of hyperlinks created by `anchor_bgn()`.

- `handle_image(source, alt[, ismap[, align[, width[, height]]]]))`
  This method is called to handle images. The default implementation simply passes the `alt` value to the `handle_data()` method.
save_bgn()
Begins saving character data in a buffer instead of sending it to the formatter object. Retrieve the stored data via save_end(). Use of the save_bgn() / save_end() pair may not be nested.

save_end()
Ends buffering character data and returns all data saved since the preceding call to save_bgn(). If the nofill flag is false, whitespace is collapsed to single spaces. A call to this method without a preceding call to save_bgn() will raise a TypeError exception.

13.3 htmlentitydefs — Definitions of HTML general entities

This module defines a single dictionary, entitydefs, which is used by the htmllib module to provide the entitydefs member of the HTMLParser class. The definition provided here contains all the entities defined by HTML 2.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

entitydefs
A dictionary mapping HTML 2.0 entity definitions to their replacement text in ISO Latin-1.

13.4 xml.parsers.expat — Fast XML parsing using Expat

New in version 2.0.

The xml.parsers.expat module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

exception ExpatError
The exception raised when Expat reports an error.

exception error
Alias for ExpatError.

XMLParserType
The type of the return values from the ParserCreate() function.

The xml.parsers.expat module contains two functions:

ErrorString (errno)
Returns an explanatory string for a given error number errno.

ParserCreate ([encoding[, namespace_separator]])
Creates and returns a new xmlparser object. encoding, if specified, must be a string naming the encoding used by the XML data. Expat doesn’t support as many encodings as Python does, and its repertoire of encodings can’t be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If encoding is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for namespace_separator. The value must be a one-character string; a ValueError will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the
namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if namespace_separator is set to a space character (‘ ‘) and the following document is parsed:

```xml
<?xml version="1.0"?>
<root xmlns = "http://default-namespace.org/"
     xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler will receive the following strings for each element:

- http://default-namespace.org/ root
- http://www.python.org/ns/ elem1
- elem2

### 13.4.1 XMLParser Objects

xmlparser objects have the following methods:

- **Parse**(data[, isfinal])
  - Parses the contents of the string data, calling the appropriate handler functions to process the parsed data. isfinal must be true on the final call to this method. data can be the empty string at any time.

- **ParseFile**(file)
  - Parse XML data reading from the object file. file only needs to provide the read(nbytes) method, returning the empty string when there's no more data.

- **SetBase**(base)
  - Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the base argument to the ExternalEntityRefHandler, NotationDeclHandler, and UnparsedEntityDeclHandler functions.

- **GetBase**()
  - Returns a string containing the base set by a previous call to SetBase(), or None if SetBase() hasn’t been called.

- **GetInputContext**()
  - Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is None. New in version 2.1.

- **ExternalEntityParserCreate**(context[, encoding])
  - Create a “child” parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The context parameter should be the string passed to the ExternalEntityRefHandler() handler function, described below. The child parser is created with the ordered_attributes, returns_unicode and specified_attributes set to the values of this parser.

xmlparser objects have the following attributes:

- **ordered_attributes**
  - Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time. New in version 2.1.
returns_unicode
If this attribute is set to a non-zero integer, the handler functions will be passed Unicode strings. If
returns_unicode is 0, 8-bit strings containing UTF-8 encoded data will be passed to the handlers.
Changed in version 1.6: Can be changed at any time to affect the result type..

specified_attributes
If set to a non-zero integer, the parser will report only those attributes which were specified in the document
instance and not those which were derived from attribute declarations. Applications which set this need to be
especially careful to use what additional information is available from the declarations as needed to comply with
the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any

The following attributes contain values relating to the most recent error encountered by an xmlparser
object, and will only have correct values once a call to Parse() or ParseFile() has raised a
xml.parsers.expat.ExpatError exception.

ErrorByteIndex
Byte index at which an error occurred.

ErrorCode
Numeric code specifying the problem. This value can be passed to the ErrorString() function, or compared
to one of the constants defined in the errors object.

ErrorLineNumber
Line number at which an error occurred.

ErrorColumnNumber
Column number at which an error occurred.

Here is the list of handlers that can be set. To set a handler on an xmlparser object o, use o.handlername = func.
handlername must be taken from the following list, and func must be a callable object accepting the correct number
of arguments. The arguments are all strings, unless otherwise stated.

XmlDeclHandler (version, encoding, standalone)
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the appli-
cable version of the XML recommendation, the encoding of the document text, and an optional “standalone”
declaration. version and encoding will be strings of the type dictated by the returns_unicode attribute, and
standalone will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the
standalone clause was omitted. This is only available with Expat version 1.95.0 or newer. New in version 2.1.

StartDoctypeDeclHandler (doctypeName, systemId, publicId, has_internal_subset)
Called when Expat begins parsing the document type declaration (<!DOCTYPE ...). The doctypeName is
provided exactly as presented. The systemId and publicId parameters give the system and public identifiers if
specified, or None if omitted. has_internal_subset will be true if the document contains and internal document
declaration subset. This requires Expat version 1.2 or newer.

EndDoctypeDeclHandler ()
Called when Expat is done parsing the document type decleration. This requires Expat version 1.2 or newer.

ElementDeclHandler (name, model)
Called once for each element type declaration. name is the name of the element type, and model is a represen-
tation of the content model.

AttlistDeclHandler (elname, attname, type, default, required)
Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this
handler is called three times, once for each attribute. elname is the name of the element to which the declaration
applies and attname is the name of the attribute declared. The attribute type is a string passed as type; the
possible values are 'CDATA', 'ID', 'IDREF', ... default gives the default value for the attribute used when
the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values).
If the attribute is required to be given in the document instance, required will be true. This requires Expat
version 1.95.0 or newer.

**StartElementHandler** *(name, attributes)*

Called for the start of every element. *name* is a string containing the element name, and *attributes* is a dictionary mapping attribute names to their values.

**EndElementHandler** *(name)*

Called for the end of every element.

**ProcessingInstructionHandler** *(target, data)*

Called for every processing instruction.

**CharacterDataHandler** *(data)*

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the **StartCdataSectionHandler**, **EndCdataSectionHandler**, and **ElementDeclHandler** callbacks to collect the required information.

**UnparsedEntityDeclHandler** *(entityName, base, systemId, publicId, notationName)*

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use **EntityDeclHandler** instead. (The underlying function in the Expat library has been declared obsolete.)

**EntityDeclHandler** *(entityName, is parameter entity, value, base, systemId, publicId, notationName)*

Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be **None** for external entities. The *notationName* parameter will be **None** for parsed entities, and the name of the notation for unparsed entities. *is parameter entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library. New in version 2.1.

**NotationDeclHandler** *(notationName, base, systemId, publicId)*

Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be **None**.

**StartNamespaceDeclHandler** *(prefix, uri)*

Called when an element contains a namespace declaration. Namespace declarations are processed before the **StartElementHandler** is called for the element on which declarations are placed.

**EndNamespaceDeclHandler** *(prefix)*

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the **StartNamespaceDeclHandler** was called to indicate the start of each namespace declaration’s scope. Calls to this handler are made after the corresponding **EndElementHandler** for the end of the element.

**CommentHandler** *(data)*

Called for comments. *data* is the text of the comment, excluding the leading ‘<!--’ and trailing ‘-->’.

**StartCdataSectionHandler** *

Called at the start of a CDATA section. This and **StartCdataSectionHandler** are needed to be able to identify the syntactical start and end for CDATA sections.

**EndCdataSectionHandler** *

Called at the end of a CDATA section.

**DefaultValue** *(data)*

Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

**DefaultValueExpand** *(data)*

This is the same as the **DefaultValue**, but doesn’t inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

**NotStandaloneHandler** *


Called if the XML document hasn’t been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to yes in an XML declaration. If this handler returns 0, then the parser will throw an XML_ERROR_NOT_STANDALONE error. If this handler is not set, no exception is raised by the parser for this condition.

**ExternalEntityRefHandler**(context, base, systemId, publicId)
Called for references to external entities. *base* is the current base, as set by a previous call to SetBase(). The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be None. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using ExternalEntityParserCreate(*context*), initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns 0, the parser will throw an XML_ERROR_EXTERNAL_ENTITY_HANDLING error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the DefaultHandler callback, if provided.

13.4.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

**code**
Expat’s internal error number for the specific error. This will match one of the constants defined in the errors object from this module. New in version 2.1.

**lineno**
Line number on which the error was detected. The first line is numbered 1. New in version 2.1.

**offset**
Character offset into the line where the error occurred. The first column is numbered 0. New in version 2.1.

13.4.3 Example

The following program defines three handlers that just print out their arguments.

```python
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()
p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse('""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""')
```

13.4. xml.parsers.expat — Fast XML parsing using Expat
The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

### 13.4.4 Content Model Descriptions

Content modules are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content module descriptions.

The values of the first two fields are constants defined in the `model` object of the `xml.parsers.expat` module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

- **XML_CTYPENAME**
  - The element named by the model name was declared to have a content model of ANY.

- **XML_CTYPENONE**
  - The element allows a choice from a number of options; this is used for content models such as \((A \mid B \mid C)\).

- **XML_CTYPENULL**
  - The named element allows a choice from a number of options; this is used for content models such as \((A \mid B \mid C)\).

- **XML_CTYPENAME**
  - Elements which are declared to be EMPT Y have this model type.

- **XML_CTYPENAME**
  - Models which represent a series of models which follow one after the other are indicated with this model type.

  This is used for models such as \((A, B, C)\).

The constants in the quantifier group are:

- **XML_CQUANT_NONE**
  - The model is option: it can appear once or not at all, as for \(A\).

- **XML_CQUANT_OPT**
  - The model must occur one or more times (A+).

- **XML_CQUANT_MIN**
  - The model must occur zero or more times, as for \(A\).

### 13.4.5 Expat error constants

The following constants are provided in the `errors` object of the `xml.parsers.expat` module. These constants are useful in interpreting some of the attributes of the `ExpatError` exception objects raised when an error has
occurred.

The `errors` object has the following attributes:

- **XML_ERROR_ASYNC_ENTITY**
- **XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
  An entity reference in an attribute value referred to an external entity instead of an internal entity.
- **XML_ERROR_BAD_CHAR_REF**
- **XML_ERROR_BINARY_ENTITY_REF**
- **XML_ERROR_DUPLICATE_ATTRIBUTE**
  An attribute was used more than once in a start tag.
- **XML_ERROR_INCORRECT_ENCODING**
- **XML_ERROR_INVALID_TOKEN**
- **XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
  Something other than whitespace occurred after the document element.
- **XML_ERROR_MISPLACED_XML_PI**
- **XML_ERROR_NO_ELEMENTS**
  The document contains no elements.
- **XML_ERROR_NO_MEMORY**
  Expat was not able to allocate memory internally.
- **XML_ERROR_PARAM_ENTITY_REF**
- **XML_ERROR_PARTIAL_CHAR**
- **XML_ERROR_RECURSIVE_ENTITY_REF**
- **XML_ERROR_SYNTAX**
  Some unspecified syntax error was encountered.
- **XML_ERROR_TAG_MISMATCH**
  An end tag did not match the innermost open start tag.
- **XML_ERROR_UNCLOSED_TOKEN**
- **XML_ERROR_UNDEFINED_ENTITY**
  A reference was made to an entity which was not defined.
- **XML_ERROR_UNKNOWN_ENCODING**
  The document encoding is not supported by Expat.

## 13.5 `xml.dom` — The Document Object Model API

New in version 2.0.

The Document Object Model, or “DOM,” is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program’s position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead
in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build
some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard
tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or “levels” in their terminology. The Python
mapping of the API is substantially based on the DOM Level 2 recommendation. Some aspects of the API will only
become available in Python 2.1, or may only be available in particular DOM implementations.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all
by DOM Level 1, and Level 2 provides only limited improvements. There is a DOMImplementation object class
which provides access to Document creation methods, but these methods were only added in DOM Level 2 and were
not implemented in time for Python 2.0. There is also no well-defined way to access these methods without an existing
Document object. For Python 2.0, consult the documentation for each particular DOM implementation to determine
the bootstrap procedure needed to create and initialize Document and DocumentType instances.

Once you have a DOM document object, you can access the parts of your XML document through its properties and
methods. These properties are defined in the DOM specification; this portion of the reference manual describes the
interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python
mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required
(though implementations are free to support the strict mapping from IDL). See section 13.5.3, “Conformance,” for a
detailed discussion of mapping requirements.

See Also:

- Document Object Model (DOM) Level 2 Specification
  (http://www.w3.org/TR/DOM-Level-2-Core/)
  The W3C recommendation upon which the Python DOM API is based.

- Document Object Model (DOM) Level 1 Specification
  (http://www.w3.org/TR/REC-DOM-Level-1/)
  The W3C recommendation for the DOM supported by xml.dom.minidom.

- PyXML
  (http://pyxml.sourceforge.net)
  Users that require a full-featured implementation of DOM should use the PyXML package.

- CORBA Scripting with Python
  (http://cgi.omg.org/cgi-bin/doc?orbos/99-08-02.pdf)
  This specifies the mapping from OMG IDL to Python.

13.5.1 Module Contents

The xml.dom contains the following functions:

**registerDOMImplementation (name, factory)**

Register the factory function with the name name. The factory function should return an object which imple-
ments the DOMImplementation interface. The factory function can return the same object every time, or a
new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some
customization).

**getDOMImplementation (name = None, features = ())**

Return a suitable DOM implementation. The name is either well-known, the module name of a DOM imple-
mentation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation
object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this
variable is used to find the implementation.
If name is not given, consider the available implementations to find one with the required feature set. If no implementation can be found, raise an ImportError. The features list must be a sequence of (feature, version) pairs which are passed to hasFeature.

In addition, xml.dom contains the Node, and the DOM exceptions.

13.5.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Section</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMImplementation</td>
<td>13.5.2</td>
<td>Interface to the underlying implementation.</td>
</tr>
<tr>
<td>Node</td>
<td>13.5.2</td>
<td>Base interface for most objects in a document.</td>
</tr>
<tr>
<td>NodeList</td>
<td>13.5.2</td>
<td>Interface for a sequence of nodes.</td>
</tr>
<tr>
<td>DocumentType</td>
<td>13.5.2</td>
<td>Information about the declarations needed to process a document.</td>
</tr>
<tr>
<td>Document</td>
<td>13.5.2</td>
<td>Object which represents an entire document.</td>
</tr>
<tr>
<td>Element</td>
<td>13.5.2</td>
<td>Element nodes in the document hierarchy.</td>
</tr>
<tr>
<td>Attr</td>
<td>13.5.2</td>
<td>Attribute value nodes on element nodes.</td>
</tr>
<tr>
<td>Comment</td>
<td>13.5.2</td>
<td>Representation of comments in the source document.</td>
</tr>
<tr>
<td>Text</td>
<td>13.5.2</td>
<td>Nodes containing textual content from the document.</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>13.5.2</td>
<td>Processing instruction representation.</td>
</tr>
</tbody>
</table>

An additional section describes the exceptions defined for working with the DOM in Python.

DOMImplementation Objects

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

hasFeature (feature, version)

Node Objects

All of the components of an XML document are subclasses of Node.

nodeType
An integer representing the node type. Symbolic constants for the types are on the Node object: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. This is a read-only attribute.

parentNode
The parent of the current node, or None for the document node. The value is always a Node object or None. For Element nodes, this will be the parent element, except for the root element, in which case it will be the Document object. For Attr nodes, this is always None. This is a read-only attribute.

attributes
A NamedNodeList of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.
**previousSibling**

The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the *self* element’s start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be *None*. This is a read-only attribute.

**nextSibling**

The node that immediately follows this one with the same parent. See also previousSibling. If this is the last child of the parent, this attribute will be *None*. This is a read-only attribute.

**childNodes**

A list of nodes contained within this node. This is a read-only attribute.

**firstChild**

The first child of the node, if there are any, or *None*. This is a read-only attribute.

**lastChild**

The last child of the node, if there are any, or *None*. This is a read-only attribute.

**localName**

The part of the *tagName* following the colon if there is one, else the entire *tagName*. The value is a string.

**prefix**

The part of the *tagName* preceding the colon if there is one, else the empty string. The value is a string, or *None*

**namespaceURI**

The namespace associated with the element name. This will be a string or *None*. This is a read-only attribute.

**nodeName**

This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the *tagName* property for elements or the *name* property for attributes. For all node types, the value of this attribute will be either a string or *None*. This is a read-only attribute.

**nodeValue**

This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with *nodeName*. The value is a string or *None*.

**hasAttributes()**

Returns true if the node has any attributes.

**hasChildNodes()**

Returns true if the node has any child nodes.

**isEqualNode(other)**

Returns true if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

**Note:** This is based on a proposed DOM Level 3 API which is still in the “working draft” stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

**appendChild(newChild)**

Add a new child node to this node at the end of the list of children, returning *newChild*.

**insertBefore(newChild, refChild)**

Insert a new child node before an existing child. It must be the case that *refChild* is a child of this node; if not, *ValueError* is raised. *newChild* is returned.

**removeChild(oldChild)**

Remove a child node. *oldChild* must be a child of this node; if not, *ValueError* is raised. *oldChild* is returned on success. If *oldChild* will not be used further, its *unlink()* method should be called.
**replaceChild** *(newChild, oldChild)*

Replace an existing node with a new node. It must be the case that *oldChild* is a child of this node; if not, *ValueError* is raised.

**normalize()**

Join adjacent text nodes so that all stretches of text are stored as single *Text* instances. This simplifies processing text from a DOM tree for many applications. New in version 2.1.

**cloneNode**(deep)

Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

**NodeList Objects**

A *NodeList* represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: the *Element* objects provides one as its list of child nodes, and the *getElementsByTag*() and *getElementsByTagNameNS()* methods of *Node* return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

**item**(i)

Return the *i*’th item from the sequence, if there is one, or *None*. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

**length**

The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow *NodeList* objects to be used as Python sequences. All *NodeList* implementations must include support for *__len__()* and *__getitem__()*: this allows iteration over the *NodeList* in *for* statements and proper support for the *len()* built-in function.

If a DOM implementation supports modification of the document, the *NodeList* implementation must also support the *__setitem__()* and *__delitem__()* methods.

**DocumentType Objects**

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a *DocumentType* object. The *DocumentType* for a document is available from the *Document* object’s *doctype* attribute.

*DocumentType* is a specialization of *Node*, and adds the following attributes:

**publicId**

The public identifier for the external subset of the document type definition. This will be a string or *None*.

**systemId**

The system identifier for the external subset of the document type definition. This will be a URI as a string, or *None*.

**internalSubset**

A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be *None*.

**name**

The name of the root element as given in the *DOCTYPE* declaration, if present. If the was no *DOCTYPE* declaration, this will be *None*.

**entities**

This is a *NamedNodeMap* giving the definitions of external entities. For entity names defined more than once,
only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

notations
This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

Document Objects

A Document represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from Node.

documentElement
The one and only root element of the document.

createElement (tagName)
Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

createElementNS (namespaceURI, tagName)
Create and return a new element with a namespace. The tagName may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

createTextNode (data)
Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

createComment (data)
Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

createProcessingInstruction (target, data)
Create and return a processing instruction node containing the target and data passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

createAttribute (name)
Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

createAttributeNS (namespaceURI, qualifiedName)
Create and return an attribute node with a namespace. The tagName may have a prefix. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

getElementsByTagName (tagName)
Search for all descendants (direct children, children’s children, etc.) with a particular element type name.

getElementsByTagNameNS (namespaceURI, localName)
Search for all descendants (direct children, children’s children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

Element Objects

Element is a subclass of Node, so inherits all the attributes of that class.

tagName
The element type name. In a namespace-using document it may have colons in it. The value is a string.

`getElementsByTagName(tagName)`
Same as equivalent method in the **Document** class.

`getElementsByTagNameNS(tagName)`
Same as equivalent method in the **Document** class.

`getAttribute(attname)`
Return an attribute value as a string.

`getAttributeNode(attname)`
Return the **Attr** node for the attribute named by `attname`.

`getAttributeNS(namespaceURI, localName)`
Return an attribute value as a string, given a `namespaceURI` and `localName`.

`getAttributeNodeNS(namespaceURI, localName)`
Return an attribute value as a node, given a `namespaceURI` and `localName`.

`removeAttribute(attname)`
Remove an attribute by name. No exception is raised if there is no matching attribute.

`removeAttributeNode(oldAttr)`
Remove and return `oldAttr` from the attribute list, if present. If `oldAttr` is not present, `NotFoundError` is raised.

`removeAttributeNS(namespaceURI, localName)`
Remove an attribute by name. Note that it uses a `localName`, not a qname. No exception is raised if there is no matching attribute.

`setAttribute(attname, value)`
Set an attribute value from a string.

`setAttributeNode(newAttr)`
Add a new attribute node to the element, replacing an existing attribute if necessary if the `name` attribute matches. If a replacement occurs, the old attribute node will be returned. If `newAttr` is already in use, `InuseAttributeErr` will be raised.

`setAttributeNodeNS(newAttr)`
Add a new attribute node to the element, replacing an existing attribute if necessary if the `namespaceURI` and `localName` attributes match. If a replacement occurs, the old attribute node will be returned. If `newAttr` is already in use, `InuseAttributeErr` will be raised.

`setAttributeNS(namespaceURI, qname, value)`
Set an attribute value from a string, given a `namespaceURI` and a `qname`. Note that a `qname` is the whole attribute name. This is different than above.

Attr Objects

**Attr** inherits from **Node**, so inherits all its attributes.

`name`
The attribute name. In a namespace-using document it may have colons in it.

`localName`
The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

`prefix`
The part of the name preceding the colon if there is one, else the empty string.
NamedNodeMap Objects

NamedNodeMap does not inherit from Node.

length
   The length of the attribute list.

item(index)
   Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the value attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized getAttribute*()-family methods on the Element objects.

Comment Objects

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

data
   The content of the comment as a string. The attribute contains all characters between the leading <!-- and trailing -->, but does not include them.

Text and CDATASection Objects

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM’s XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide different values for the nodeType attribute.

These interfaces extend the Node interface. They cannot have child nodes.

data
   The content of the text node as a string.

Note: The use of a CDATASection node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent CDATASection nodes represent different CDATA marked sections.

ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the Node interface and cannot have child nodes.

target
   The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

data
   The content of the processing instruction following the first whitespace character.

Exceptions

New in version 2.1.

The DOM Level 2 recommendation defines a single exception, DOMException, and a number of constants that allow applications to determine what sort of error occurred. DOMException instances carry a code attribute that provides the appropriate value for the specific exception.
The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the `code` attribute.

**exception DOMException**  
Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

**exception DomstringSizeErr**  
Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

**exception HierarchyRequestErr**  
Raised when an attempt is made to insert a node where the node type is not allowed.

**exception IndexSizeErr**  
Raised when an index or size parameter to a method is negative or exceeds the allowed values.

**exception InuseAttributeErr**  
Raised when an attempt is made to insert an `Attr` node that is already present elsewhere in the document.

**exception InvalidAccessErr**  
Raised if a parameter or an operation is not supported on the underlying object.

**exception InvalidCharacterErr**  
This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

**exception InvalidModificationErr**  
Raised when an attempt is made to modify the type of a node.

**exception InvalidStateErr**  
Raised when an attempt is made to modify the type of an object that is not or is no longer usable.

**exception NamespaceErr**  
If an attempt is made to change any object in a way that is not permitted with regard to the *Namespaces in XML* recommendation, this exception is raised.

**exception NotFoundErr**  
Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

**exception NotSupportedErr**  
Raised when the implementation does not support the requested type of object or operation.

**exception NoDataAllowedErr**  
This is raised if data is specified for a node which does not support data.

**exception NoModificationAllowedErr**  
Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

**exception SyntaxErr**  
Raised when an invalid or illegal string is specified.

**exception WrongDocumentErr**  
Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:
13.5.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The primitive IDL types used in the DOM specification are mapped to Python types according to the following table.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>IntegerType (with a value of 0 or 1)</td>
</tr>
<tr>
<td>int</td>
<td>IntegerType</td>
</tr>
<tr>
<td>long int</td>
<td>IntegerType</td>
</tr>
<tr>
<td>unsigned int</td>
<td>IntegerType</td>
</tr>
</tbody>
</table>

Additionally, the DOMString defined in the recommendation is mapped to a Python string or Unicode string. Applications should be able to handle Unicode whenever a string is returned from the DOM.

The IDL null value is mapped to None, which may be accepted or provided by the implementation whenever null is allowed by the API.

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL attribute declarations in much the way the Java mapping does. Mapping the IDL declarations

```python
readonly attribute string someValue;
attribute string anotherValue;
```

yields three accessor functions: a “get” method for `someValue(_get_someValue())`, and “get” and “set” methods for `anotherValue(_get_anotherValue() and _set_anotherValue())`. The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is not required to work, and may raise an AttributeError.
The Python DOM API, however, does require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

Additionally, the accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. “Set” accessors should never be provided for `readonly` attributes.

## 13.6 xml.dom.minidom — Lightweight DOM implementation

New in version 2.0.

`xml.dom.minidom` is a light-weight implementation of the Document Object Model interface. It is intended to be simpler than the full DOM and also significantly smaller.

DOM applications typically start by parsing some XML into a DOM. With `xml.dom.minidom`, this is done through the parse functions:

```python
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\temp\mydata.xml')  # parse an XML file by name

datasource = open('c:\temp\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')</n```

The parse function can take either a filename or an open file object.

**parse** *(filename_or_file, parser)*

Return a `Document` from the given input. `filename_or_file` may be either a file name, or a file-like object. `parser`, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead:

**parseString** *(string[, parser]*)

Return a `Document` that represents the `string`. This method creates a `StringIO` object for the string and passes that on to `parse`.

Both functions return a `Document` object representing the content of the document.

You can also create a `Document` node merely by instantiating a document object. Then you could add child nodes to it to populate the DOM:

```python
from xml.dom.minidom import Document

newdoc = Document()
newel = newdoc.createElement("some_tag")
newdoc.appendChild(newel)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the
**documentElement** property. It gives you the main element in the XML document: the one that holds all others.

Here is an example program:

```python
dom3 = parseString("<myxml>Some data</myxml>"")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM, you should clean it up. This is necessary because some versions of Python do not support garbage collection of objects that refer to each other in a cycle. Until this restriction is removed from all versions of Python, it is safest to write your code as if cycles would not be cleaned up.

The way to clean up a DOM is to call its `unlink()` method:

```python
dom1.unlink()
dom2.unlink()
dom3.unlink()
```

`unlink()` is a `xml.dom.minidom`-specific extension to the DOM API. After calling `unlink()` on a node, the node and its descendents are essentially useless.

**See Also:**

Document Object Model (DOM) Level 1 Specification
(http://www.w3.org/TR/REC-DOM-Level-1/)

The W3C recommendation for the DOM supported by `xml.dom.minidom`.

### 13.6.1 DOM objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

**unlink()**

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the `Document` object, but may be called on child nodes to discard children of that node.

**writexml**(writer)

Write XML to the writer object. The writer should have a `write()` method which matches that of the file object interface.

**toxml()**

Return the XML that the DOM represents as a string.

The following standard DOM methods have special considerations with `xml.dom.minidom`:

**cloneNode**(deep)

Although this method was present in the version of `xml.dom.minidom` packaged with Python 2.0, it was seriously broken. This has been corrected for subsequent releases.

### 13.6.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.
import xml.dom.minidom

document = """
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>
<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"

dom = xml.dom.minidom.parseString(document)

space = " ">
def getText(nodelist):
    rc = ""
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc = rc + node.data
    return rc

def handleSlideshow(slideshow):
    print "<html>
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>"
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

dom = xml.dom.minidom.parseString(document)
13.6.3 minidom and the DOM standard

The xml.dom.minidom module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the Document object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.

- Operations are used as methods. Since the DOM uses only in parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. void operations return None.

- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute foo can also be accessed through accessor methods _get_foo() and _set_foo(). readonly attributes must not be changed; this is not enforced at runtime.

- The types short int, unsigned int, unsigned long long, and boolean all map to Python integer objects.

- The type DOMString maps to Python strings. xml.dom.minidom supports either byte or Unicode strings, but will normally produce Unicode strings. Attributes of type DOMString may also be None.

- const declarations map to variables in their respective scope (e.g. xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE); they must not be changed.

- DOMException is currently not supported in xml.dom.minidom. Instead, xml.dom.minidom uses standard Python exceptions such as TypeError and AttributeError.

- NodeList objects are implemented as Python’s built-in list type, so don’t support the official API, but are much more “Pythonic.”

The following interfaces have no implementation in xml.dom.minidom:

- DOMTimeStamp
- DocumentType (added in Python 2.1)
- DOMImplementation (added in Python 2.1)
- CharacterData
- CDATASection
- Notation
- Entity
- EntityReference
- DocumentFragment

Most of these reflect information in the XML document that is not of general utility to most DOM users.
13.7 xml.dom.pulldom — Support for building partial DOM trees

New in version 2.0.
xml.dom.pulldom allows building only selected portions of a Document Object Model representation of a docu-
ment from SAX events.

class PullDOM([documentFactory])
    xml.sax.handler.ContentHandler implementation that ...}

class DOMEventStream(stream, parser, bufsize)
...

class SAX2DOM([documentFactory])
    xml.sax.handler.ContentHandler implementation that ...

parse(stream_or_string[, parser[, bufsize]])
...

parseString(string[, parser])
...

default_bufsize
    Default value for the bufsize parameter to parse(). Changed in version 2.1: The value of this variable can be
    changed before calling parse() and the new value will take effect.

13.7.1 DOMEventStream Objects

getEvent()
...

expandNode(node)
...

reset()
...

13.8 xml.sax — Support for SAX2 parsers

New in version 2.0.
The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface
for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used
by users of the SAX API.
The convenience functions are:

make_parser([parser_list])
    Create and return a SAX XMLReader object. The first parser found will be used. If parser_list is provided, it
must be a sequence of strings which name modules that have a function named create_parser(). Modules
listed in parser_list will be used before modules in the default list of parsers.

parse(filename_or_stream, handler[, error_handler])
    Create a SAX parser and use it to parse a document. The document, passed in as filename_or_stream, can
be a filename or a file object. The handler parameter needs to be a SAX ContentHandler instance. If error_handler
is given, it must be a SAX ErrorHandler instance; if omitted, SAXParseException will
be raised on all errors. There is no return value; all work must be done by the handler passed in.
parseString (string, handler[, error_handler])

Similar to parse(), but parses from a buffer string received as a parameter.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. “Reader” in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The InputSource, Locator, AttributesImpl, AttributesNSImpl, and XMLReader interfaces are defined in the module xml.sax.xmlreader. The handler interfaces are defined in xml.sax.handler. For convenience, InputSource (which is often instantiated directly) and the handler classes are also available from xml.sax. These interfaces are described below.

In addition to these classes, xml.sax provides the following exception classes.

exception SAXException (msg[, exception])

Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the ErrorHandler interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, msg should be a human-readable description of the error. The optional exception parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception SAXParseException (msg, exception, locator)

Subclass of SAXException raised on parse errors. Instances of this class are passed to the methods of the SAX ErrorHandler interface to provide information about the parse error. This class supports the SAX Locator interface as well as the SAXException interface.

exception SAXNotRecognizedException (msg[, exception])

Subclass of SAXException raised when a SAX XMLReader is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception SAXNotSupportedException (msg[, exception])

Subclass of SAXException raised when a SAX XMLReader is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See Also:

SAX: The Simple API for XML
(http://www.megginsong.com/SAX/)

This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

13.8.1 SAXException Objects

The SAXException exception class supports the following methods:

getMessage ()

Return a human-readable message describing the error condition.
getException()

Return an encapsulated exception object, or None.

13.9 xml.sax.handler — Base classes for SAX handlers

New in version 2.0.

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module xml.sax, so that all methods get default implementations.

class ContentHandler()

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class DTDHandler()

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class EntityResolver()

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class ErrorHandler()

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, xml.sax.handler provides symbolic constants for the feature and property names.

feature_namespaces

Value: "http://xml.org/sax/features/namespaces"

true: Perform Namespace processing (default).
false: Optionally do not perform Namespace processing (implies namespace-prefixes).
access: (parsing) read-only; (not parsing) read/write

feature_namespace_prefixes

Value: "http://xml.org/sax/features/namespace-prefixes"

true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

feature_string_interning

Value: "http://xml.org/sax/features/string-interning"

true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

feature_validation

Value: "http://xml.org/sax/features/validation"

true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

feature_external_ges

Value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

**feature_external_pes**
Value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

**all_features**
List of all features.

**property_lexical_handler**
Value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

**property_declaration_handler**
Value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

**property_dom_node**
Value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

**property_xml_string**
Value: "http://xml.org/sax/properties/xml-string"
data type: String
description: The literal string of characters that was the source for the current event.
access: read-only

**all_properties**
List of all known property names.

### 13.9.1 ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document:

**setDocumentLocator (locator)**
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application’s business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.
startDocument()
    Receive notification of the beginning of a document.
    The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler
    (except for setDocumentLocator()).

endDocument()
    Receive notification of the end of a document.
    The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The
    parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or
    reached the end of input.

startPrefixMapping(prefix, uri)
    Begin the scope of a prefix-URI Namespace mapping.
    The information from this event is not necessary for normal Namespace processing: the SAX
    XML reader will automatically replace prefixes for element and attribute names when the
    http://xml.org/sax/features/namespaces feature is true (the default).
    There are cases, however, when applications need to use prefixes in character data or in attribute values, where
    they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the
    application to expand prefixes in those contexts itself, if necessary.
    Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other:
    all startPrefixMapping() events will occur before the corresponding startElement event, and all
    endPrefixMapping() events will occur after the corresponding endElement() event, but their order
    is not guaranteed.

endPrefixMapping(prefix)
    End the scope of a prefix-URI mapping.
    See startPrefixMapping() for details. This event will always occur after the corresponding endElement
    event, but the order of endPrefixMapping events is not otherwise guaranteed.

startElement(name, attrs)
    Signals the start of an element in non-namespace mode.
    The name parameter contains the raw XML 1.0 name of the element type as a string and the attrs parameter
    holds an instance of the Attributes class containing the attributes of the element.

endElement(name)
    Signals the end of an element in non-namespace mode.
    The name parameter contains the name of the element type, just as with the startElement event.

startElementNS(name, qname, attrs)
    Signals the start of an element in namespace mode.
    The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter
    the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the
    AttributesNS class containing the attributes of the element.
    Parsers may set the qname parameter to None, unless the http://xml.org/sax/features/namespace-prefixes
    feature is activated.

endElementNS(name, qname)
    Signals the end of an element in namespace mode.
    The name parameter contains the name of the element type, just as with the startElementNS event, likewise the
    qname parameter.

characters(content)
    Receive notification of character data.
The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.  

\textit{content} may be a Unicode string or a byte string; the \texttt{expat} reader module produces always Unicode strings.  

\textbf{Note:} The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use \textit{content} instead of slicing \textit{content} with the old \textit{offset} and \textit{length} parameters.

\begin{verbatim}
ignoreWhitespace()

Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

processingInstruction(target, data)

Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

skippedEntity(name)

Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the \texttt{http://xml.org/sax/features/external-general-entities} and the \texttt{http://xml.org/sax/features/external-parameter-entities} properties.

\end{verbatim}

13.9.2 DTDHandler Objects

\texttt{DTDHandler} instances provide the following methods:

\begin{verbatim}
notationDecl(name, publicId, systemId)

Handle a notation declaration event.

unparsedEntityDecl(name, publicId, systemId, ndata)

Handle an unparsed entity declaration event.

\end{verbatim}

13.9.3 EntityResolver Objects

\begin{verbatim}
resolveEntity(publicId, systemId)

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns \textit{systemId}.

\end{verbatim}

13.9.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the \texttt{XMLReader}. If you create an object that implements this interface, then register the object with your \texttt{XMLReader}, the parser will call the methods
in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a SAXParseException as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

**error** *(exception)*
Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

**fatalError** *(exception)*
Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

**warning** *(exception)*
Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

### 13.10 `xml.sax.saxutils` — SAX Utilities

New in version 2.0.

The module `xml.sax.saxutils` contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

**escape** *(data[, entities]*)
Escape &amp;, ¡, and ¿ in a string of data.
You can escape other strings of data by passing a dictionary as the optional entities parameter. The keys and values must all be strings; each key will be replaced with its corresponding value.

**XMLGenerator** *(out[, encoding]*)
This class implements the `ContentHandler` interface by writing SAX events back into an XML document. In other words, using an `XMLGenerator` as the content handler will reproduce the original document being parsed. `out` should be a file-like object which will default to `sys.stdout`. `encoding` is the encoding of the output stream which defaults to 'iso-8859-1'.

**XMLFilterBase** *(base)*
This class is designed to sit between an `XMLReader` and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

**prepare_input_source** *(source[, base]*)
This function takes an input source and an optional base URL and returns a fully resolved InputSource object ready for reading. The input source can be given as a string, a file-like object, or an `InputSource` object; parsers will use this function to implement the polymorphic `source` argument to their `parse()` method.

### 13.11 `xml.sax.xmlreader` — Interface for XML parsers

New in version 2.0.

SAX parsers implement the `XMLReader` interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by `xml.sax.make_parser()` with no arguments to create a new parser object.

**XMLReader** *
Base class which can be inherited by SAX parsers.
class IncrementalParser()

In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still parse() won’t return until the entire document is processed. So these interfaces should be used if the blocking behaviour of parse() is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must not be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class Locator()

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return None.

class InputSource( [ systemId ] )

Encapsulation of the information needed by the XMLReader to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the XMLReader.parse() method and for returning from EntityResolver.resolveEntity.

An InputSource belongs to the application, the XMLReader is not allowed to modify InputSource objects passed to it from the application, although it may make copies and modify those.

class AttributesImpl( attrs )

This is a dictionary-like object which represents the element attributes in a startElement() call. In addition to the most useful dictionary operations, it supports a number of other methods as described below. Objects of this class should be instantiated by readers; attrs must be a dictionary-like object.

class AttributesNSImpl( attrs, qnames )

Namespace-aware variant of attributes, which will be passed to startElementNS(). It is derived from AttributesImpl, but understands attribute names as two-tuples of namespaceURI and localname. In addition, it provides a number of methods expecting qualified names as they appear in the original document.

13.11.1 XMLReader Objects

The XMLReader interface supports the following methods:

parse( source )

Process an input source, producing SAX events. The source object can be a system identifier (i.e. a string identifying the input source – typically a file name or an URL), a file-like object, or an InputSource object. When parse() returns, the input is completely processed, and the parser object can be discarded or reset. As a limitation, the current implementation only accepts byte streams; processing of character streams is for further study.

getContentHandler()

Return the current ContentHandler.

setContentTypeHandler( handler )

Set the current ContentHandler. If no ContentHandler is set, content events will be discarded.

getDTDHandler()

Return the current DTDHandler.
setDTDHandler \(\text{(handler)}\)
Set the current DTDHandler. If no DTDHandler is set, DTD events will be discarded.

getEntityResolver()
Return the current EntityResolver.

setEntityResolver \(\text{(handler)}\)
Set the current EntityResolver. If no EntityResolver is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

getErrorHandler()
Return the current ErrorHandler.

setErrorHandler \(\text{(handler)}\)
Set the current error handler. If no ErrorHandler is set, errors will be raised as exceptions, and warnings will be printed.

setLocale \(\text{(locale)}\)
Allow an application to set the locale for errors and warnings.
SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must throw a SAX exception. Applications may request a locale change in the middle of a parse.

getFeature \(\text{(featurename)}\)
Return the current setting for feature \(\text{featurename}\). If the feature is not recognized, SAXNotRecognizedException is raised. The well-known featurenames are listed in the module xml.sax.handler.

setFeature \(\text{(featurename, value)}\)
Set the \(\text{featurename}\) to \(\text{value}\). If the feature is not recognized, SAXNotSupportedException is raised. If the feature or its setting is not supported by the parser, SAXNotSupportedException is raised.

getProperty \(\text{(propertyname)}\)
Return the current setting for property \(\text{propertyname}\). If the property is not recognized, a SAXNotRecognizedException is raised. The well-known propertynames are listed in the module xml.sax.handler.

setProperty \(\text{(propertyname, value)}\)
Set the \(\text{propertyname}\) to \(\text{value}\). If the property is not recognized, SAXNotRecognizedException is raised. If the property or its setting is not supported by the parser, SAXNotSupportedException is raised.

13.11.2 IncrementalParser Objects

Instances of IncrementalParser offer the following additional methods:

feed \(\text{(data)}\)
Process a chunk of \(\text{data}\).

close()
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

reset()
This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.”
13.11.3 Locator Objects

Instances of Locator provide these methods:

getColumnName()  
Return the column number where the current event ends.

getAddress()  
Return the line number where the current event ends.

getPublicId()  
Return the public identifier for the current event.

getSystemId()  
Return the system identifier for the current event.

13.11.4 InputSource Objects

setPublicId(id)  
Sets the public identifier of this InputSource.

getPublicId()  
Returns the public identifier of this InputSource.

setSystemId(id)  
Sets the system identifier of this InputSource.

getSystemId()  
Returns the system identifier of this InputSource.

setEncoding(encoding)  
Sets the character encoding of this InputSource.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the InputSource is ignored if the InputSource also contains a character stream.

getEncoding()  
Get the character encoding of this InputSource.

setByteStream(bytefile)  
Set the byte stream (a Python file-like object which does not perform byte-to-character conversion) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

getByteStream()  
Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or None if unknown.

setCharacterStream(charfile)  
Set the character stream for this input source. (The stream must be a Python 1.6 Unicode-wrapped file-like that performs conversion to Unicode strings.)

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

getCharacterStream()  
Get the character stream for this input source.
13.11.5 AttributesImpl Objects

AttributesImpl objects implement a portion of the mapping protocol, and the methods copy(), get(), has_key(), items(), keys(), and values(). The following methods are also provided:

getLength()
Return the number of attributes.

getNames()
Return the names of the attributes.

getType(name)
Returns the type of the attribute name, which is normally ‘CDATA’.

getValue(name)
Return the value of attribute name.

13.11.6 AttributesNSImpl Objects

getValueByQName(name)
Return the value for a qualified name.

getNameByQName(name)
Return the (namespace, localname) pair for a qualified name.

getQNameByName(name)
Return the qualified name for a (namespace, localname) pair.

getQNames()
Return the qualified names of all attributes.

13.12 xmllib — A parser for XML documents

Deprecated since release 2.0. Use xml.sax instead. The newer XML package includes full support for XML 1.0.

Changed in version 1.5.2: Added namespace support.

This module defines a class XMLParser which serves as the basis for parsing text files formatted in XML (Extensible Markup Language).

class XMLParser()
The XMLParser class must be instantiated without arguments.¹

This class provides the following interface methods and instance variables:

attributes
A mapping of element names to mappings. The latter mapping maps attribute names that are valid for the element to the default value of the attribute, or if there is no default to None. The default value is the empty dictionary. This variable is meant to be overridden, not extended since the default is shared by all instances of XMLParser.

¹Actually, a number of keyword arguments are recognized which influence the parser to accept certain non-standard constructs. The following keyword arguments are currently recognized. The defaults for all of these is 0 (false) except for the last one for which the default is 1 (true). accept_unquoted_attributes (accept certain attribute values without requiring quotes), accept_missing_endtag_name (accept end tags that look like </>), map_case (map upper case to lower case in tags and attributes), accept_utf8 (allow UTF-8 characters in input; this is required according to the XML standard, but Python does not as yet deal properly with these characters, so this is not the default), translate_attribute_references (don’t attempt to translate character and entity references in attribute values).
elements
A mapping of element names to tuples. The tuples contain a function for handling the start and end tag re-
spectively of the element, or None if the method unknown_starttag() or unknown_endtag() is to be
called. The default value is the empty dictionary. This variable is meant to be overridden, not extended since
the default is shared by all instances of XMLParser.

entitydefs
A mapping of entitynames to their values. The default value contains definitions for 'lt', 'gt', 'amp',
'quot', and 'apos'.

reset()
Reset the instance. Loses all unprocessed data. This is called implicitly at the instantiation time.

setnomoretags()
Stop processing tags. Treat all following input as literal input (CDATA).

setliteral()
Enter literal mode (CDATA mode). This mode is automatically exited when the close tag matching the last
unclosed open tag is encountered.

feed(data)
Feed some text to the parser. It is processed insofar as it consists of complete tags; incomplete data is buffered
until more data is fed or close() is called.

close()
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined
by a derived class to define additional processing at the end of the input, but the redefined version should always
call close().

translate_references(data)
Translate all entity and character references in data and return the translated string.

getnamespace()
Return a mapping of namespace abbreviations to namespace URIs that are currently in effect.

handle_xml(encoding, standalone)
This method is called when the '<?xml ...?>' tag is processed. The arguments are the values of the en-
coding and standalone attributes in the tag. Both encoding and standalone are optional. The values passed to
handle_xml() default to None and the string 'no' respectively.

handle_doctype(tag, pubid, syslit, data)
This method is called when the '<!DOCTYPE...>' declaration is processed. The arguments are the tag name
of the root element, the Formal Public Identifier (or None if not specified), the system identifier, and the uninter-
preted contents of the internal DTD subset as a string (or None if not present).

handle_starttag(tag, method, attributes)
This method is called to handle start tags for which a start tag handler is defined in the instance variable
elements. The tag argument is the name of the tag, and the method argument is the function (method)
which should be used to support semantic interpretation of the start tag. The attributes argument is a dictio-
nary of attributes, the key being the name and the value being the value of the attribute found inside the tag’s
<> brackets. Character and entity references in the value have been interpreted. For instance, for the start
tag <A HREF="http://www.cwi.nl/">, this method would be called as handle_starttag('A',
self.elements['A'][0], {'HREF': 'http://www.cwi.nl/'}). The base implementation
simply calls method with attributes as the only argument.

handle_endtag(tag, method)
This method is called to handle endtags for which an end tag handler is defined in the instance variable
elements. The tag argument is the name of the tag, and the method argument is the function (method)
which should be used to support semantic interpretation of the end tag. For instance, for the endtag</A>, this
method would be called as handle_endtag('A', self.elements['A'][1]). The base implemen-
The `xmlparse` class provides a number of methods for handling different kinds of data.

**handle_data**(data)
This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

**handle_charref**(ref)
This method is called to process a character reference of the form `&#ref;`. `ref` can either be a decimal number, or a hexadecimal number when preceded by an `x`. In the base implementation, `ref` must be a number in the range 0-255. It translates the character to ASCII and calls the method `handle_data()` with the character as argument. If `ref` is invalid or out of range, the method `unknown_charref()` is called to handle the error. A subclass must override this method to provide support for character references outside of the ASCII range.

**handle_comment**(comment)
This method is called when a comment is encountered. The `comment` argument is a string containing the text between the `<!--` and `-->` delimiters, but not the delimiters themselves. For example, the comment `<!--text-->` will cause this method to be called with the argument `text`. The default method does nothing.

**handle_cdata**(data)
This method is called when a CDATA element is encountered. The `data` argument is a string containing the text between the `<![CDATA[` and `]]>` delimiters, but not the delimiters themselves. For example, the entity `<![CDATA[text]]>` will cause this method to be called with the argument `text`. The default method does nothing, and is intended to be overridden.

**handle_proc**(name, data)
This method is called when a processing instruction (PI) is encountered. The `name` is the PI target, and the `data` argument is a string containing the text between the PI target and the closing delimiter, but not the delimiter itself. For example, the instruction `<?XML text?>` will cause this method to be called with the arguments `XML` and `text`. The default method does nothing. Note that if a document starts with `<?xml ..?>`, `handle_xml()` is called to handle it.

**handle_special**(data)
This method is called when a declaration is encountered. The `data` argument is a string containing the text between the `<!` and `>` delimiters, but not the delimiters themselves. For example, the entity declaration `<!ENTITY text>` will cause this method to be called with the argument `ENTITY text`. The default method does nothing. Note that `<!DOCTYPE ..>` is handled separately if it is located at the start of the document.

**syntax_error**(message)
This method is called when a syntax error is encountered. The `message` is a description of what was wrong. The default method raises a `RuntimeError` exception. If this method is overridden, it is permissible for it to return. This method is only called when the error can be recovered from. Unrecoverable errors raise a `RuntimeError` without first calling `syntax_error()`.

**unknown_starttag**(tag, attributes)
This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_endtag**(tag)
This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_charref**(ref)
This method is called to process unresolvable numeric character references. It is intended to be overridden by a derived class; the base class implementation does nothing.

**unknown_entityref**(ref)
This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation calls `syntax_error()` to signal an error.
See Also:

Extensible Markup Language (XML) 1.0
(http://www.w3.org/TR/REC-xml)

The XML specification, published by the World Wide Web Consortium (W3C), defines the syntax and processor requirements for XML. References to additional material on XML, including translations of the specification, are available at http://www.w3.org/XML/.

Python and XML Processing
(http://www.python.org/topics/xml/)

The Python XML Topic Guide provides a great deal of information on using XML from Python and links to other sources of information on XML.

SIG for XML Processing in Python
(http://www.python.org/sigs/xml-sig/)

The Python XML Special Interest Group is developing substantial support for processing XML from Python.

13.12.1 XML Namespaces

This module has support for XML namespaces as defined in the XML Namespaces proposed recommendation.

Tag and attribute names that are defined in an XML namespace are handled as if the name of the tag or element consisted of the namespace (i.e. the URL that defines the namespace) followed by a space and the name of the tag or attribute. For instance, the tag <html xmlns='http://www.w3.org/TR/REC-html40'> is treated as if the tag name was ‘http://www.w3.org/TR/REC-html40 html’, and the tag <html:a href='http://frob.com'> inside the above mentioned element is treated as if the tag name were ‘http://www.w3.org/TR/REC-html40 a’ and the attribute name as if it were ‘http://www.w3.org/TR/REC-html40 href’.

An older draft of the XML Namespaces proposal is also recognized, but triggers a warning.

See Also:

Namespaces in XML
(http://www.w3.org/TR/REC-xml-names/)

This World-Wide Web Consortium recommendation describes the proper syntax and processing requirements for namespaces in XML.
Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here’s an overview:

- **audioop** — Manipulate raw audio data.
- **imageop** — Manipulate raw image data.
- **aifc** — Read and write audio files in AIFF or AIFC format.
- **sunau** — Provide an interface to the Sun AU sound format.
- **wave** — Provide an interface to the WAV sound format.
- **chunk** — Module to read IFF chunks.
- **colorsys** — Conversion functions between RGB and other color systems.
- **rgbimg** — Read and write image files in “SGI RGB” format (the module is not SGI specific though!).
- **imghdr** — Determine the type of image contained in a file or byte stream.
- **sndhdr** — Determine type of a sound file.

### 14.1 audioop — Manipulate raw audio data

The audioop module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the al and sunaudiodev modules. All scalar items are integers, unless specified otherwise.

This module provides support for u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

- **exception error**
  This exception is raised on all errors, such as unknown number of bytes per sample, etc.

- **add** *(fragment1, fragment2, width)*
  Return a fragment which is the addition of the two samples passed as parameters. width is the sample width in bytes, either 1, 2 or 4. Both fragments should have the same length.

- **adpcm2lin** *(adpcmfragment, width, state)*
  Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of lin2adpcm() for details on ADPCM coding. Return a tuple *(sample, newstate)* where the sample has the width specified in width.

- **adpcm32lin** *(adpcmfragment, width, state)*
  Decode an alternative 3-bit ADPCM code. See lin2adpcm3() for details.

- **avg** *(fragment, width)*
  Return the average over all samples in the fragment.
avgpp \((\text{fragment, width})\)
Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

bias \((\text{fragment, width, bias})\)
Return a fragment that is the original fragment with a bias added to each sample.

cross \((\text{fragment, width})\)
Return the number of zero crossings in the fragment passed as an argument.

findfactor \((\text{fragment, reference})\)
Return a factor \(F\) such that \(\text{rms} \left( \text{add} \left( \text{fragment, mul} \left( \text{reference, -F} \right) \right) \right)\) is minimal, i.e., return the factor with which you should multiply \(\text{reference}\) to make it match as well as possible to \(\text{fragment}\). The fragments should both contain 2-byte samples.
The time taken by this routine is proportional to \(\text{len} \left( \text{fragment} \right)\).

findfit \((\text{fragment, reference})\)
Try to match \(\text{reference}\) as well as possible to a portion of \(\text{fragment}\) (which should be the longer fragment). This is (conceptually) done by taking slices out of \(\text{fragment}\), using \text{findfactor()} to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple \((\text{offset, factor})\) where \(\text{offset}\) is the (integer) offset into \(\text{fragment}\) where the optimal match started and \(\text{factor}\) is the (floating-point) factor as per \text{findfactor()}.

findmax \((\text{fragment, length})\)
Search \(\text{fragment}\) for a slice of length \(\text{length}\) samples (not bytes!) with maximum energy, i.e., return \(i\) for which \(\text{rms} \left( \text{fragment}[i*2:(i+\text{length})*2] \right)\) is maximal. The fragments should both contain 2-byte samples.
The routine takes time proportional to \(\text{len} \left( \text{fragment} \right)\).

getsample \((\text{fragment, width, index})\)
Return the value of sample \(\text{index}\) from the fragment.

lin2lin \((\text{fragment, width, newwidth})\)
Convert samples between 1-, 2- and 4-byte formats.

lin2adpcm \((\text{fragment, width, state})\)
Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.
\(\text{state}\) is a tuple containing the state of the coder. The coder returns a tuple \((\text{adpcmfrag, newstate})\), and the \text{newstate} should be passed to the next call of \text{lin2adpcm()}. In the initial call, \text{None} can be passed as the \text{state}. \text{adpcmfrag} is the ADPCM coded fragment packed 2 4-bit values per byte.

lin2adpcm3 \((\text{fragment, width, state})\)
This is an alternative ADPCM coder that uses only 3 bits per sample. It is not compatible with the Intel/DVI ADPCM coder and its output is not packed (due to laziness on the side of the author). Its use is discouraged.

lin2ulaw \((\text{fragment, width})\)
Convert samples in the audio fragment to u-LAW encoding and return this as a Python string. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

minmax \((\text{fragment, width})\)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

max \((\text{fragment, width})\)
Return the maximum of the absolute value of all samples in a fragment.

maxpp \((\text{fragment, width})\)
Return the maximum peak-peak value in the sound fragment.
**mul** *(fragment, width, factor)*
Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Overflow is silently ignored.

**ratecv** *(fragment, width, nchannels, inrate, outrage, state[ , weightA[ , weightB ] ] )*
Convert the frame rate of the input fragment.
*state* is a tuple containing the state of the converter. The converter returns a tuple *(newfragment, newstate)*, and *newstate* should be passed to the next call of *ratecv()*.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to 1 and 0 respectively.

**reverse** *(fragment, width)*
Reverse the samples in a fragment and returns the modified fragment.

**rms** *(fragment, width)*
Return the root-mean-square of the fragment, i.e.

\[ \sqrt{\frac{\sum S_i^2}{n}} \]

This is a measure of the power in an audio signal.

**tomono** *(fragment, width, lfactor, rfactor)*
Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

**tostereo** *(fragment, width, lfactor, rfactor)*
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

**ulaw2lin** *(fragment, width)*
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as *mul()* or *max()* make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```python
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(sample, width, lfactor)
    rsample = audioop.mul(sample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to *lin2adpcm()* along to the decoder, not the final state (as returned by the coder). If you want to use *struct.struct()* to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The *find*() routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample...
and subtract the whole output sample from the input sample:

```python
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)  # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    # out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata,2,-factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

### 14.2 imageop — Manipulate raw image data

The `imageop` module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in Python strings. This is the same format as used by `gl.lrectwrite()` and the `imgfile` module.

The module defines the following variables and functions:

- **exception error**
  - This exception is raised on all errors, such as unknown number of bits per pixel, etc.

- **crop**
  - `image, psize, width, height, x0, y0, x1, y1`
  - Return the selected part of `image`, which should by `width` by `height` in size and consist of pixels of `psize` bytes. `x0`, `y0`, `x1` and `y1` are like the `gl.lrectread()` parameters, i.e., the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If `x0` is bigger than `x1` the new image is mirrored. The same holds for the `y` coordinates.

- **scale**
  - `image, psize, width, height, newwidth, newheight`
  - Return `image` scaled to size `newwidth` by `newheight`. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

- **tovideo**
  - `image, psize, width, height`
  - Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

- **grey2mono**
  - `image, width, height, threshold`
  - Convert a 8-bit deep greyscale image to a 1-bit deep image by thresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to `mono2grey()`.

- **dither2mono**
  - `image, width, height`
  - Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

- **mono2grey**
  - `image, width, height, p0, p1`
  - Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value `p0` on output and all one-value input pixels get value `p1` on output. To convert a monochrome black-and-white image to greyscale pass the values 0 and 255 respectively.

- **grey2grey4**
  - `image, width, height`
  - Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.
**grey2grey2** \((image, \ width, \ height)\)
Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

**dither2grey2** \((image, \ width, \ height)\)
Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for **dither2mono()**, the dithering algorithm is currently very simple.

**grey42grey** \((image, \ width, \ height)\)
Convert a 4-bit greyscale image to an 8-bit greyscale image.

**grey22grey** \((image, \ width, \ height)\)
Convert a 2-bit greyscale image to an 8-bit greyscale image.

### 14.3 aifc — Read and write AIFF and AIFC files

This module provides support for reading and writing AIFF and AIF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIF-C is a newer version of the format that includes the ability to compress the audio data.

**Caveat:** Some operations may only work under IRIX; these will raise `ImportError` when attempting to import the `cl` module, which is only available on IRIX.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of \(n\text{channels} \times \text{sampsize}\) bytes, and a second's worth of audio consists of \(n\text{channels} \times \text{sampsize} \times \text{framerate}\) bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes, i.e. 176,400 bytes.

Module `aifc` defines the following function:

**open** \((file[, \ mode ])\)
Open an AIFF or AIF-C file and return an object instance with methods that are described below. The argument `file` is either a string naming a file or a file object. `mode` must be ‘r’ or ‘rb’ when the file must be opened for reading, or ‘w’ or ‘wb’ when the file must be opened for writing. If omitted, `file.mode` is used if it exists, otherwise ‘rb’ is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`.

Objects returned by `open()` when a file is opened for reading have the following methods:

**getnchannels()**
Return the number of audio channels (1 for mono, 2 for stereo).

**getsampwidth()**
Return the size in bytes of individual samples.

**getframerate()**
Return the sampling rate (number of audio frames per second).

**getnframes()**
Return the number of audio frames in the file.

**getcomptype()**
Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is ‘NONE’.

**getcompname()**
Return a human-readable description of the type of compression used in the audio file. For AIFF files, the
The returned value is ‘not compressed’.

**getparams()**
Return a tuple consisting of all of the above values in the above order.

**getmarkers()**
Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

**getmark(id)**
Return the tuple as described in **getmarkers()** for the mark with the given id.

**readframes(nframes)**
Read and return the next nframes frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

**rewind()**
Rewind the read pointer. The next **readframes()** will start from the beginning.

**setpos(pos)**
Seek to the specified frame number.

**tell()**
Return the current frame number.

**close()**
Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by **open()** when a file is opened for writing have all the above methods, except for **readframes()** and **setpos()**. In addition the following methods exist. The **get*()** methods can only be called after the corresponding **set*()** methods have been called. Before the first **writeframes()** or **writeframesraw()**, all parameters except for the number of frames must be filled in.

**aiff()**
Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in ‘.aiff’ in which case the default is an AIFF file.

**aifc()**
Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in ‘.aiff’ in which case the default is an AIFF file.

**setnchannels(nchannels)**
Specify the number of channels in the audio file.

**setsampwidth(width)**
Specify the size in bytes of audio samples.

**setframerate(rate)**
Specify the sampling frequency in frames per second.

**setnframes(nframes)**
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

**setcomptype(type, name)**
Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

**setparams(nchannels, sampwidth, framerate, comptype, compname)**
Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means
that it is possible to use the result of a `getparams()` call as argument to `setparams()`.

**setmark** (*id, pos, name*)

Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before `close()`.

**tell**()

Return the current write position in the output file. Useful in combination with `setmark()`.

**writeframes** (*data*)

Write data to the output file. This method can only be called after the audio file parameters have been set.

**writeframesraw** (*data*)

Like `writeframes()`, except that the header of the audio file is not updated.

**close**()

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

### 14.4 sunau — Read and write Sun AU files

The `sunau` module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules `aifc` and `wave`.

An audio file consists of a header followed by the data. The fields of the header are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic word</td>
<td>The four bytes <code>.snd</code></td>
</tr>
<tr>
<td>header size</td>
<td>Size of the header, including info, in bytes.</td>
</tr>
<tr>
<td>data size</td>
<td>Physical size of the data, in bytes.</td>
</tr>
<tr>
<td>encoding</td>
<td>Indicates how the audio samples are encoded.</td>
</tr>
<tr>
<td>sample rate</td>
<td>The sampling rate.</td>
</tr>
<tr>
<td># of channels</td>
<td>The number of channels in the samples.</td>
</tr>
<tr>
<td>info</td>
<td>ASCII string giving a description of the audio file (padded with null bytes).</td>
</tr>
</tbody>
</table>

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The `sunau` module defines the following functions:

**open** (*file, mode*)

If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

- `'r'` Read only mode.
- `'w'` Write only mode.

Note that it does not allow read/write files.

A *mode* of `'r'` returns a `AU_read` object, while a *mode* of `'w'` or `'wb'` returns a `AU_write` object.

**openfp** (*file, mode*)

A synonym for `open`, maintained for backwards compatibility.

The `sunau` module defines the following exception:

**exception Error**

An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The `sunau` module defines the following data items:
**AUDIO_FILE_MAGIC**

An integer every valid Sun AU file begins with, stored in big-endian form. This is the string `.snd` interpreted as an integer.

**AUDIO_FILE_ENCODING_MULAW_8**
**AUDIO_FILE_ENCODING_LINEAR_8**
**AUDIO_FILE_ENCODING_LINEAR_16**
**AUDIO_FILE_ENCODING_LINEAR_24**
**AUDIO_FILE_ENCODING_LINEAR_32**
**AUDIO_FILE_ENCODING_ALAW_8**

Values of the encoding field from the AU header which are supported by this module.

**AUDIO_FILE_ENCODING_FLOAT**
**AUDIO_FILE_ENCODING_DOUBLE**
**AUDIO_FILE_ENCODING_ADPCM_G721**
**AUDIO_FILE_ENCODING_ADPCM_G722**
**AUDIO_FILE_ENCODING_ADPCM_G723_3**
**AUDIO_FILE_ENCODING_ADPCM_G723_5**

Additional known values of the encoding field from the AU header, but which are not supported by this module.

### 14.4.1 AU_read Objects

AU_read objects, as returned by `open()` above, have the following methods:

**close()**
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

**getnchannels()**
Returns number of audio channels (1 for mono, 2 for stereo).

**getsampwidth()**
Returns sample width in bytes.

**getframerate()**
Returns sampling frequency.

**getnframes()**
Returns number of audio frames.

**getcomptype()**
Returns compression type. Supported compression types are `'ULAW'`, `'ALAW'` and `'NONE'`.

**getcompname()**
Human-readable version of `getcomptype()`. The supported types have the respective names `'CCITT G.711 u-law'`, `'CCITT G.711 A-law'` and `'not compressed'`.

**getparams()**
Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

**readframes(n)**
Reads and returns at most `n` frames of audio, as a string of bytes. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

**rewind()**
Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.
setpos \((pos)\)
Set the file pointer to the specified position. Only values returned from \(tell()\) should be used for \(pos\).

tell ()
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the \texttt{aifc}, and don’t do anything interesting.

getmarkers ()
Returns \texttt{None}.

getmark \((id)\)
Raise an error.

### 14.4.2 \texttt{AU\_write Objects}

\texttt{AU\_write} objects, as returned by \texttt{open()} above, have the following methods:

setnchannels \((n)\)
Set the number of channels.

setsampwidth \((n)\)
Set the sample width (in bytes.)

setframerate \((n)\)
Set the frame rate.

setnframes \((n)\)
Set the number of frames. This can be later changed, when and if more frames are written.

setcomptype \((type, name)\)
Set the compression type and description. Only \texttt{’NONE’} and \texttt{’ULAW’} are supported on output.

setparams \((tuple)\)
The \texttt{tuple} should be \((nchannels, sampwidth, framerate, nframes, comptype, compname)\), with values valid for the set\_\*() methods. Set all parameters.

tell ()
Return current position in the file, with the same disclaimer for the \texttt{AU\_read.tell()} and \texttt{AU\_read.setpos()} methods.

writeframesraw \((data)\)
Write audio frames, without correcting \texttt{nframes}.

writeframes \((data)\)
Write audio frames and make sure \texttt{nframes} is correct.

close ()
Make sure \texttt{nframes} is correct, and close the file.
This method is called upon deletion.

Note that it is invalid to set any parameters after calling \texttt{writeframes()} or \texttt{writeframesraw()}.

### 14.5 \texttt{wave} — Read and write WAV files

The \texttt{wave} module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The \texttt{wave} module defines the following function and exception:
**open** *(file[, mode]*)

If *file* is a string, open the file by that name, other treat it as a seekable file-like object. *mode* can be any of

‘r’, ‘rb’ | Read only mode.
---|---
‘w’, ‘wb’ | Write only mode.

Note that it does not allow read/write WAV files.

A *mode* of ‘r’ or ‘rb’ returns a *Wave_read* object, while a *mode* of ‘w’ or ‘wb’ returns a *Wave_write* object. If *mode* is omitted and a file-like object is passed as *file*, *file*.mode is used as the default value for *mode* (the ‘b’ flag is still added if necessary).

**openfp** *(file, mode)*

A synonym for **open()**, maintained for backwards compatibility.

**exception** *Error*

An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

### 14.5.1 Wave_read Objects

Wave_read objects, as returned by **open()**, have the following methods:

**close** *

Close the stream, and make the instance unusable. This is called automatically on object collection.

**getnchannels** *

Returns number of audio channels (1 for mono, 2 for stereo).

**getsampwidth** *

Returns sample width in bytes.

**getframerate** *

Returns sampling frequency.

**getnframes** *

Returns number of audio frames.

**getcomptype** *

Returns compression type (‘NONE’ is the only supported type).

**getcompname** *

Human-readable version of **getcomptype()**. Usually ‘not compressed’ parallels ‘NONE’.

**getparams** *

Returns a tuple (*nchannels*, *sampwidth*, *framerate*, *nframes*, *comptype*, *compname*), equivalent to output of the **get*()** methods.

**readframes** *(n)*

Reads and returns at most *n* frames of audio, as a string of bytes.

**rewind** *

Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the **aifi** module, and don’t do anything interesting.

**getmarkers** *

Returns None.

**getmark** *(id)*

Raise an error.
The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

\[ \textit{setpos}(\textit{pos}) \]
Set the file pointer to the specified position.

\[ \textit{tell}() \]
Return current file pointer position.

14.5.2 Wave_write Objects

Wave_write objects, as returned by \texttt{open()}, have the following methods:

\[ \textit{close}() \]
Make sure \textit{nframes} is correct, and close the file. This method is called upon deletion.

\[ \textit{setnchannels}(\textit{n}) \]
Set the number of channels.

\[ \textit{setsampwidth}(\textit{n}) \]
Set the sample width to \textit{n} bytes.

\[ \textit{setframerate}(\textit{n}) \]
Set the frame rate to \textit{n}.

\[ \textit{setnframes}(\textit{n}) \]
Set the number of frames to \textit{n}. This will be changed later if more frames are written.

\[ \textit{setcomptype}(\textit{type, name}) \]
Set the compression type and description.

\[ \textit{setparams}(\textit{tuple}) \]
The \textit{tuple} should be \texttt{(nchannels, sampwidth, framerate, nframes, comptype, compname)}, with values valid for the \textit{set*()} methods. Sets all parameters.

\[ \textit{tell}() \]
Return current position in the file, with the same disclaimer for the Wave_read.tell() and Wave_read.setpos() methods.

\[ \textit{writeframesraw}(\textit{data}) \]
Write audio frames, without correcting \textit{nframes}.

\[ \textit{writeframes}(\textit{data}) \]
Write audio frames and make sure \textit{nframes} is correct.

Note that it is invalid to set any parameters after calling \textit{writeframes()} or \textit{writeframesraw()}, and any attempt to do so will raise \texttt{wave.Error}.

14.6 chunk — Read IFF chunked data

This module provides an interface for reading files that use EA IFF 85 chunks. This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

\[ ^1 \text{“EA IFF 85” Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.} \]
The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the Chunk class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with a EOFError exception.

```python
class Chunk(file[, align, bigendian, inclheader])
```

Class which represents a chunk. The file argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is read(). If the methods seek() and tell() are present and don’t raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument align is true, chunks are assumed to be aligned on 2-byte boundaries. If align is false, no alignment is assumed. The default value is true. If the optional argument bigendian is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument inclheader is true, the size given in the chunk header includes the size of the header. The default value is false.

A Chunk object supports the following methods:

- `getname()`
  Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

- `getsize()`
  Returns the size of the chunk.

- `close()`
  Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise IOError if called after the close() method has been called.

- `isatty()`
  Returns 0.

- `seek(pos[, whence])`
  Set the chunk’s current position. The whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

- `tell()`
  Return the current position into the chunk.

- `read([size])`
  Read at most size bytes from the chunk (less if the read hits the end of the chunk before obtaining size bytes). If the size argument is negative or omitted, read all data until the end of the chunk. The bytes are returned as a string object. An empty string is returned when the end of the chunk is encountered immediately.

- `skip()`
  Skip to the end of the chunk. All further calls to read() for the chunk will return ‘’. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

### 14.7 colorsys — Conversions between color systems
The colorsys module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

More information about color spaces can be found at http://www.inforamp.net/%7epoynton/ColorFAQ.html.

The colorsys module defines the following functions:

- `rgb_to_yiq(r, g, b)`: Convert the color from RGB coordinates to YIQ coordinates.
- `yiq_to_rgb(y, i, q)`: Convert the color from YIQ coordinates to RGB coordinates.
- `rgb_to_hls(r, g, b)`: Convert the color from RGB coordinates to HLS coordinates.
- `hls_to_rgb(h, l, s)`: Convert the color from HLS coordinates to RGB coordinates.
- `rgb_to_hsv(r, g, b)`: Convert the color from RGB coordinates to HSV coordinates.
- `hsv_to_rgb(h, s, v)`: Convert the color from HSV coordinates to RGB coordinates.

Example:

```python
>>> import colorsys
>>> colorsys.rgb_to_hsv(.3, .4, .2)
(0.25, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.25, 0.5, 0.4)
(0.3, 0.4, 0.2)
```

14.8 rgbimg — Read and write “SGI RGB” files

The rgbimg module allows Python programs to access SGI imglib image files (also known as ‘.rgb’ files). The module is far from complete, but is provided anyway since the functionality that there is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

- `exception error`: This exception is raised on all errors, such as unsupported file type, etc.
- `sizeofimage(file)`: This function returns a tuple `(x, y)` where `x` and `y` are the size of the image in pixels. Only 4 byte RGBA pixels, 3 byte RGB pixels, and 1 byte greyscale pixels are currently supported.
- `longimagedata(file)`: This function reads and decodes the image on the specified file, and returns it as a Python string. The string has 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite()`, for instance.
- `longstoimage(data, x, y, z, file)`: This function writes the RGBA data in `data` to image file `file`. `x` and `y` give the size of the image. `z` is 1 if the
saved image should be 1 byte greyscale, 3 if the saved image should be 3 byte RGB data, or 4 if the saved images should be 4 byte RGBA data. The input data always contains 4 bytes per pixel. These are the formats returned by `gl.lrectread()`.

**ttob (flag)**

This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.

### 14.9 `imghdr` — Determine the type of an image

The `imghdr` module determines the type of image contained in a file or byte stream.

The `imghdr` module defines the following function:

**what (filename[, h])**

Tests the image data contained in the file named by `filename`, and returns a string describing the image type. If optional `h` is provided, the `filename` is ignored and `h` is assumed to contain the byte stream to test.

The following image types are recognized, as listed below with the return value from `what()`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Image format</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rgb'</td>
<td>SGI ImgLib Files</td>
</tr>
<tr>
<td>'gif'</td>
<td>GIF 87a and 89a Files</td>
</tr>
<tr>
<td>'pnm'</td>
<td>Portable Graymap Files</td>
</tr>
<tr>
<td>'pgm'</td>
<td>Portable Pixmap Files</td>
</tr>
<tr>
<td>'ppm'</td>
<td>Portable Pixmap Files</td>
</tr>
<tr>
<td>'tiff'</td>
<td>TIFF Files</td>
</tr>
<tr>
<td>'rast'</td>
<td>Sun Raster Files</td>
</tr>
<tr>
<td>'xbm'</td>
<td>X Bitmap Files</td>
</tr>
<tr>
<td>'jpeg'</td>
<td>JPEG data in JFIF format</td>
</tr>
<tr>
<td>'bmp'</td>
<td>BMP files</td>
</tr>
<tr>
<td>'png'</td>
<td>Portable Network Graphics</td>
</tr>
</tbody>
</table>

You can extend the list of file types `imghdr` can recognize by appending to this variable:

**tests**

A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When `what()` is called with a byte-stream, the file-like object will be `None`.

The test function should return a string describing the image type if the test succeeded, or `None` if it failed.

Example:

```python
>>> import imghdr
>>> imghdr.what('/tmp/bass.gif')
'gif'
```

### 14.10 `sndhdr` — Determine type of sound file

The `sndhdr` provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a tuple `(type, sampling_rate,`
channels, frames, bits_per_sample). The value for type indicates the data type and will be one of the strings 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul'. The sampling_rate will be either the actual value or 0 if unknown or difficult to decode. Similarly, channels will be either the number of channels or 0 if it cannot be determined or if the value is difficult to decode. The value for frames will be either the number of frames or -1. The last item in the tuple, bits_per_sample, will either be the sample size in bits or 'A' for A-LAW or 'U' for u-LAW.

what (filename)
Determines the type of sound data stored in the file filename using whathdr(). If it succeeds, returns a tuple as described above, otherwise None is returned.

whathdr (filename)
Determines the type of sound data stored in a file based on the file header. The name of the file is given by filename. This function returns a tuple as described above on success, or None.
Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here’s an overview:

- **md5**  
  RSA’s MD5 message digest algorithm.  
- **sha**  
  NIST’s secure hash algorithm, SHA.  
- **mpz**  
  Interface to the GNU MP library for arbitrary precision arithmetic.  
- **rotor**  
  Enigma-like encryption and decryption.

Hardcore cypherpunks will probably find the cryptographic modules written by Andrew Kuchling of further interest; the package adds built-in modules for DES and IDEA encryption, provides a Python module for reading and decrypting PGP files, and then some. These modules are not distributed with Python but available separately. See the URL http://starship.python.net/crew/amk/python/code/crypto.html or send email to amk1@bigfoot.com for more information.

15.1 md5 — MD5 message digest algorithm

This module implements the interface to RSA’s MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use `new()` to create an md5 object. You can now feed this object with arbitrary strings using the `update()` method, and at any point you can ask it for the `digest` (a strong kind of 128-bit checksum, a.k.a. “fingerprint”) of the concatenation of the strings fed to it so far using the `digest()` method.

For example, to obtain the digest of the string ‘Nobody inspects the spammish repetition’:

```python
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\x8d\xdf0\xff\xe9'
```

More condensed:

```python
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\x8d\xdf0\xff\xe9'
```

`new([arg])`

Return a new md5 object. If `arg` is present, the method call `update(arg)` is made.

`md5([arg])`
An md5 object has the following methods:

**update (arg)**
Update the md5 object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments, i.e. m.update(a); m.update(b) is equivalent to m.update(a+b).

**digest ()**
Return the digest of the strings passed to the update() method so far. This is a 16-byte string which may contain non-ASCII characters, including null bytes.

**hexdigest ()**
Like digest() except the digest is returned as a string of length 32, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**copy ()**
Return a copy (“clone”) of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

**See Also:**
Module sha (section 15.2):
Similar module implementing the Secure Hash Algorithm (SHA). The SHA algorithm is considered a more secure hash.

### 15.2 sha — SHA message digest algorithm

This module implements the interface to NIST’s secure hash algorithm, known as SHA. It is used in the same way as the md5 module: use new() to create an sha object, then feed this object with arbitrary strings using the update() method, and at any point you can ask it for the digest of the concatenation of the strings fed to it so far. SHA digests are 160 bits instead of MD5’s 128 bits.

**new (string)**
Return a new sha object. If string is present, the method call update(string) is made.

The following values are provided as constants in the module and as attributes of the sha objects returned by new():

**blocksize**
Size of the blocks fed into the hash function; this is always 1. This size is used to allow an arbitrary string to be hashed.

**digestsize**
The size of the resulting digest in bytes. This is always 20.

An sha object has the same methods as md5 objects:

**update (arg)**
Update the sha object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments, i.e. m.update(a); m.update(b) is equivalent to m.update(a+b).

**digest ()**
Return the digest of the strings passed to the update() method so far. This is a 20-byte string which may contain non-ASCII characters, including null bytes.

**hexdigest ()**
Like digest() except the digest is returned as a string of length 40, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**copy ()**
Return a copy (“clone”) of the sha object. This can be used to efficiently compute the digests of strings that
share a common initial substring.

See Also:

Secure Hash Standard
(http://csrc.nist.gov/fips/fip180-1.txt)

The Secure Hash Algorithm is defined by NIST document FIPS PUB 180-1: Secure Hash Standard, published in April of 1995. It is available online as plain text (at least one diagram was omitted) and as PDF at http://csrc.nist.gov/fips/fip180-1.pdf.

15.3  mpz — GNU arbitrary magnitude integers

This is an optional module. It is only available when Python is configured to include it, which requires that the GNU MP software is installed.

This module implements the interface to part of the GNU MP library, which defines arbitrary precision integer and rational number arithmetic routines. Only the interfaces to the integer (mpz_*) routines are provided. If not stated otherwise, the description in the GNU MP documentation can be applied.

Support for rational numbers can be implemented in Python. For an example, see the Rat module, provided as ‘Demos/classes/Rat.py’ in the Python source distribution.

In general, mpz-numbers can be used just like other standard Python numbers, e.g., you can use the built-in operators like +, *, etc., as well as the standard built-in functions like abs(), int(), ..., divmod(), pow(). Please note: the bitwise-xor operation has been implemented as a bunch of ands, inverts and ors, because the library lacks an mpz_xor() function, and I didn’t need one.

You create an mpz-number by calling the function mpz() (see below for an exact description). An mpz-number is printed like this: mpz(value).

```
mpz (value)
```

Create a new mpz-number. value can be an integer, a long, another mpz-number, or even a string. If it is a string, it is interpreted as an array of radix-256 digits, least significant digit first, resulting in a positive number. See also the binary() method, described below.

**MPZType**

The type of the objects returned by mpz() and most other functions in this module.

A number of extra functions are defined in this module. Non mpz-arguments are converted to mpz-values first, and the functions return mpz-numbers.

```
powm (base, exponent, modulus)
```

Return pow(base, exponent) % modulus. If exponent == 0, return mpz(1). In contrast to the C library function, this version can handle negative exponents.

```
gcd (op1, op2)
```

Return the greatest common divisor of op1 and op2.

```
gcdext (a, b)
```

Return a tuple (g, s, t), such that a*s + b*t == g == gcd(a, b).

```
sqrt (op)
```

Return the square root of op. The result is rounded towards zero.

```
sqrtrcm (op)
```

Return a tuple (root, remainder), such that root*root + remainder == op.

```
divm (numerator, denominator, modulus)
```

Returns a number q such that q * denominator % modulus == numerator. One could also implement this function in Python, using gcdext().

15.3.  mpz — GNU arbitrary magnitude integers 409
An mpz-number has one method:

binary()
Convert this mpz-number to a binary string, where the number has been stored as an array of radix-256 digits, least significant digit first.

The mpz-number must have a value greater than or equal to zero, otherwise ValueError will be raised.

15.4 rotor — Enigma-like encryption and decryption

This module implements a rotor-based encryption algorithm, contributed by Lance Ellinghouse. The design is derived from the Enigma device, a machine used during World War II to encipher messages. A rotor is simply a permutation. For example, if the character ‘A’ is the origin of the rotor, then a given rotor might map ‘A’ to ‘L’, ‘B’ to ‘Z’, ‘C’ to ‘G’, and so on. To encrypt, we choose several different rotors, and set the origins of the rotors to known positions; their initial position is the ciphering key. To encipher a character, we permute the original character by the first rotor, and then apply the second rotor’s permutation to the result. We continue until we’ve applied all the rotors; the resulting character is our ciphertext. We then change the origin of the final rotor by one position, from ‘A’ to ‘B’; if the final rotor has made a complete revolution, then we rotate the next-to-last rotor by one position, and apply the same procedure recursively. In other words, after enciphering one character, we advance the rotors in the same fashion as a car’s odometer. Decoding works in the same way, except we reverse the permutations and apply them in the opposite order.

The available functions in this module are:

newrotor (key[, numrotors])
Return a rotor object. key is a string containing the encryption key for the object; it can contain arbitrary binary data. The key will be used to randomly generate the rotor permutations and their initial positions. numrotors is the number of rotor permutations in the returned object; if it is omitted, a default value of 6 will be used.

Rotor objects have the following methods:

setkey (key)
Sets the rotor’s key to key.

encrypt (plaintext)
Reset the rotor object to its initial state and encrypt plaintext, returning a string containing the ciphertext. The ciphertext is always the same length as the original plaintext.

encryptmore (plaintext)
Encrypt plaintext without resetting the rotor object, and return a string containing the ciphertext.

decrypt (ciphertext)
Reset the rotor object to its initial state and decrypt ciphertext, returning a string containing the plaintext. The plaintext string will always be the same length as the ciphertext.

decryptmore (ciphertext)
Decrypt ciphertext without resetting the rotor object, and return a string containing the plaintext.

An example usage:
The module’s code is not an exact simulation of the original Enigma device; it implements the rotor encryption scheme differently from the original. The most important difference is that in the original Enigma, there were only 5 or 6 different rotors in existence, and they were applied twice to each character; the cipher key was the order in which they were placed in the machine. The Python rotor module uses the supplied key to initialize a random number generator; the rotor permutations and their initial positions are then randomly generated. The original device only enciphered the letters of the alphabet, while this module can handle any 8-bit binary data; it also produces binary output. This module can also operate with an arbitrary number of rotors.

The original Enigma cipher was broken in 1944. The version implemented here is probably a good deal more difficult to crack (especially if you use many rotors), but it won’t be impossible for a truly skillful and determined attacker to break the cipher. So if you want to keep the NSA out of your files, this rotor cipher may well be unsafe, but for discouraging casual snooping through your files, it will probably be just fine, and may be somewhat safer than using the UNIX crypt command.
Restricted Execution

In general, Python programs have complete access to the underlying operating system through the various functions and classes. For example, a Python program can open any file for reading and writing by using the `open()` built-in function (provided the underlying OS gives you permission!). This is exactly what you want for most applications.

There exists a class of applications for which this “openness” is inappropriate. Take Grail: a web browser that accepts “applets,” snippets of Python code, from anywhere on the Internet for execution on the local system. This can be used to improve the user interface of forms, for instance. Since the originator of the code is unknown, it is obvious that it cannot be trusted with the full resources of the local machine.

**Restricted execution** is the basic framework in Python that allows for the segregation of trusted and untrusted code. It is based on the notion that trusted Python code (a *supervisor*) can create a “padded cell” (or environment) with limited permissions, and run the untrusted code within this cell. The untrusted code cannot break out of its cell, and can only interact with sensitive system resources through interfaces defined and managed by the trusted code. The term “restricted execution” is favored over “safe-Python” since true safety is hard to define, and is determined by the way the restricted environment is created. Note that the restricted environments can be nested, with inner cells creating subcells of lesser, but never greater, privilege.

An interesting aspect of Python’s restricted execution model is that the interfaces presented to untrusted code usually have the same names as those presented to trusted code. Therefore no special interfaces need to be learned to write code designed to run in a restricted environment. And because the exact nature of the padded cell is determined by the supervisor, different restrictions can be imposed, depending on the application. For example, it might be deemed “safe” for untrusted code to read any file within a specified directory, but never to write a file. In this case, the supervisor may redefine the built-in `open()` function so that it raises an exception whenever the `mode` parameter is `'w'`. It might also perform a `chroot()`-like operation on the `filename` parameter, such that root is always relative to some safe “sandbox” area of the filesystem. In this case, the untrusted code would still see a built-in `open()` function in its environment, with the same calling interface. The semantics would be identical too, with `IOErrors` being raised when the supervisor determined that an unallowable parameter is being used.

The Python run-time determines whether a particular code block is executing in restricted execution mode based on the identity of the `__builtins__` object in its global variables: if this is (the dictionary of) the standard `__builtin__` module, the code is deemed to be unrestricted, else it is deemed to be restricted.

Python code executing in restricted mode faces a number of limitations that are designed to prevent it from escaping from the padded cell. For instance, the function object attribute `func_globals` and the class and instance object attribute `__dict__` are unavailable.

Two modules provide the framework for setting up restricted execution environments:

- `rexec`: Basic restricted execution framework.
- `Bastion`: Providing restricted access to objects.

See Also:

Grail, an Internet browser written in Python, is available at http://grail.cnri.reston.va.us/grail/. More information on the use of Python’s restricted execution mode in Grail is available on the Web site.

16.1 rexec — Restricted execution framework

This module contains the RExec class, which supports \_\_eval\_\_(), \_\_execfile\_\_(), \_\_exec\_\_(), and \_\_import\_\_() methods, which are restricted versions of the standard Python functions eval(), execfile() and the exec and import statements. Code executed in this restricted environment will only have access to modules and functions that are deemed safe; you can subclass RExec to add or remove capabilities as desired.

Note: The RExec class can prevent code from performing unsafe operations like reading or writing disk files, or using TCP/IP sockets. However, it does not protect against code using extremely large amounts of memory or CPU time.

class RExec([hooks[, verbose]])

Returns an instance of the RExec class.

hooks is an instance of the RHooks class or a subclass of it. If it is omitted or None, the default RHooks class is instantiated. Whenever the rexec module searches for a module (even a built-in one) or reads a module’s code, it doesn’t actually go out to the file system itself. Rather, it calls methods of an RHooks instance that was passed to or created by its constructor. (Actually, the RExec object doesn’t make these calls — they are made by a module loader object that’s part of the RExec object. This allows another level of flexibility, e.g. using packages.)

By providing an alternate RHooks object, we can control the file system accesses made to import a module, without changing the actual algorithm that controls the order in which those accesses are made. For instance, we could substitute an RHooks object that passes all filesystem requests to a file server elsewhere, via some RPC mechanism such as ILU. Grail’s applet loader uses this to support importing applets from a URL for a directory.

If verbose is true, additional debugging output may be sent to standard output.

The RExec class has the following class attributes, which are used by the \_\_init\_\_() method. Changing them on an existing instance won’t have any effect; instead, create a subclass of RExec and assign them new values in the class definition. Instances of the new class will then use those new values. All these attributes are tuples of strings.

nok\_builtin\_names

Contains the names of built-in functions which will not be available to programs running in the restricted environment. The value for RExec is (‘open’, ‘reload’, ‘\_\_import\_\_’). (This gives the exceptions, because by far the majority of built-in functions are harmless. A subclass that wants to override this variable should probably start with the value from the base class and concatenate additional forbidden functions — when new dangerous built-in functions are added to Python, they will also be added to this module.)

ok\_builtin\_modules

A similar remark about overriding this variable applies — use the value from the base class as a starting point.

ok\_path

Contains the directories which will be searched when an import is performed in the restricted environment. The value for RExec is the same as sys.path (at the time the module is loaded) for unrestricted code.

ok\_posix\_names


ok\_sys\_names

Contains the names of the functions and variables in the sys module which will be available to pro-
grams running in the restricted environment. The value for RExec is ('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint').

RExec instances support the following methods:

**r_eval** *(code)*

code must either be a string containing a Python expression, or a compiled code object, which will be evaluated in the restricted environment’s __main__ module. The value of the expression or code object will be returned.

**r_exec** *(code)*

code must either be a string containing one or more lines of Python code, or a compiled code object, which will be executed in the restricted environment’s __main__ module.

**r_execfile** *(filename)*

Execute the Python code contained in the file filename in the restricted environment’s __main__ module.

Methods whose names begin with ‘s_’ are similar to the functions beginning with ‘r_’, but the code will be granted access to restricted versions of the standard I/O streams sys.stdin, sys.stderr, and sys.stdout.

**s_eval** *(code)*

code must be a string containing a Python expression, which will be evaluated in the restricted environment.

**s_exec** *(code)*

code must be a string containing one or more lines of Python code, which will be executed in the restricted environment.

**s_execfile** *(code)*

Execute the Python code contained in the file filename in the restricted environment.

RExec objects must also support various methods which will be implicitly called by code executing in the restricted environment. Overriding these methods in a subclass is used to change the policies enforced by a restricted environment.

**r_import** *(modulename[, globals[, locals[, fromlist]]]])*

Import the module modulename, raising an ImportError exception if the module is considered unsafe.

**r_open** *(filename[, mode[, bufsize]]]])*

Method called when open() is called in the restricted environment. The arguments are identical to those of open(), and a file object (or a class instance compatible with file objects) should be returned. RExec’s default behaviour is allow opening any file for reading, but forbidding any attempt to write a file. See the example below for an implementation of a less restrictive r_open().

**r_reload** *(module)*

Reload the module object module, re-parsing and re-initializing it.

**r_unload** *(module)*

Unload the module object module (i.e., remove it from the restricted environment’s sys.modules dictionary).

And their equivalents with access to restricted standard I/O streams:

**s_import** *(modulename[, globals[, locals[, fromlist]]]])*

Import the module modulename, raising an ImportError exception if the module is considered unsafe.

**s_reload** *(module)*

Reload the module object module, re-parsing and re-initializing it.

**s_unload** *(module)*

Unload the module object module.
16.1.1 An example

Let us say that we want a slightly more relaxed policy than the standard RExec class. For example, if we’re willing to allow files in ‘/tmp’ to be written, we can subclass the RExec class:

```python
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # check filename: must begin with /tmp/
            if file[:5]!='/tmp/':
                raise IOError, "can’t write outside /tmp"
            elif (string.find(file, '/../') >= 0 or
                  file[:3] == '../' or file[-3:] == '/..'):
                raise IOError, "'..' in filename forbidden"
        else: raise IOError, "Illegal open() mode"
        return open(file, mode, buf)
```

Notice that the above code will occasionally forbid a perfectly valid filename: for example, code in the restricted environment won’t be able to open a file called ‘/tmp/foo/../bar’. To fix this, the r_open() method would have to simplify the filename to ‘/tmp/bar’, which would require splitting apart the filename and performing various operations on it. In cases where security is at stake, it may be preferable to write simple code which is sometimes overly restrictive, instead of more general code that is also more complex and may harbor a subtle security hole.

16.2 Bastion — Restricting access to objects

According to the dictionary, a bastion is “a fortified area or position”, or “something that is considered a stronghold.” It’s a suitable name for this module, which provides a way to forbid access to certain attributes of an object. It must always be used with the rexec module, in order to allow restricted-mode programs access to certain safe attributes of an object, while denying access to other, unsafe attributes.

```python
class BastionClass(getfunc, name)
    Class which actually implements bastion objects. This is the default class used by Bastion(). The getfunc parameter is a function which returns the value of an attribute which should be exposed to the restricted execution environment when called with the name of the attribute as the only parameter. name is used to construct the repr() of the BastionClass instance.
```

Protect the object object, returning a bastion for the object. Any attempt to access one of the object’s attributes will have to be approved by the filter function; if the access is denied an AttributeError exception will be raised.

If present, filter must be a function that accepts a string containing an attribute name, and returns true if access to that attribute will be permitted; if filter returns false, the access is denied. The default filter denies access to any function beginning with an underscore (‘_’). The bastion’s string representation will be ‘<Bastion for name>’ if a value for name is provided; otherwise, ‘repr(object)’ will be used.

class, if present, should be a subclass of BastionClass; see the code in ‘bastion.py’ for the details. Overriding the default BastionClass will rarely be required.
Python Language Services

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

- **parser** — Access parse trees for Python source code.
- **symbol** — Constants representing internal nodes of the parse tree.
- **token** — Constants representing terminal nodes of the parse tree.
- **keyword** — Test whether a string is a keyword in Python.
- **tokenize** — Lexical scanner for Python source code.
- **tabnanny** — Tool for detecting white space related problems in Python source files in a directory tree.
- **pyclbr** — Supports information extraction for a Python class browser.
- **py_compile** — Compile Python source files to byte-code files.
- **compileall** — Tools for byte-compiling all Python source files in a directory tree.
- **dis** — Disassembler for Python byte code.

### 17.1 parser — Access Python parse trees

The parser module provides an interface to Python’s internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the parser module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to the *Python Language Reference*. The parser itself is created from a grammar specification defined in the file ‘Grammar/Grammar’ in the standard Python distribution. The parse trees stored in the AST objects created by this module are the actual output from the internal parser when created by the expr() or suite() functions, described below. The AST objects created by sequence2ast() faithfully simulate those structures. Be aware that the values of the sequences which are considered “correct” will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by ast2list() or ast2tuple() has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file.
‘include/graminit.h’ and the Python module `symbol`. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, ‘if’),` where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, ‘if’, 12),` where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file ‘Include/token.h’ and the Python module `token`.

The AST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple “wrapper” class may be created in Python to hide the use of AST objects.

The `parser` module defines functions for a few distinct purposes. The most important purposes are to create AST objects and to convert AST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an AST object.

See Also:

Module `symbol` (section 17.2):
Useful constants representing internal nodes of the parse tree.

Module `token` (section 17.3):
Useful constants representing leaf nodes of the parse tree and functions for testing node values.

17.1.1 Creating AST Objects

AST objects may be created from source code or from a parse tree. When creating an AST object from source, different functions are used to create the ‘eval’ and ‘exec’ forms.

`expr(source)`
The `expr()` function parses the parameter `source` as if it were an input to `compile(source, ‘file.py’, ‘eval’)`. If the parse succeeds, an AST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

`suite(source)`
The `suite()` function parses the parameter `source` as if it were an input to `compile(source, ‘file.py’, ‘exec’)`. If the parse succeeds, an AST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

`sequence2ast(sequence)`
This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an AST object is created from the internal representation and returned to the caller. If there is a problem creating the internal representation, or if the tree cannot be validated, a `ParserError` exception is thrown. An AST object created this way should not be assumed to compile correctly; normal exceptions thrown by compilation may still be initiated when the AST object is passed to `compileast()`. This may indicate problems not related to syntax (such as a `MemoryError` exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, ‘name’) or as three-element lists of the form `(1, ‘name’, 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in
the input tree.

**tuple2ast** (*sequence*)

This is the same function as `sequence2ast()`. This entry point is maintained for backward compatibility.

### 17.1.2 Converting AST Objects

AST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

**ast2list** (*ast*, *line_info*)

This function accepts an AST object from the caller in `ast` and returns a Python list representing the equivalent parse tree. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, `ast2tuple()` should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

**ast2tuple** (*ast*, *line_info*)

This function accepts an AST object from the caller in `ast` and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to `ast2list()`.

If `line_info` is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

**compileast** (*ast*, *filename* = `<ast>`)

The Python byte compiler can be invoked on an AST object to produce code objects which can be used as part of an `exec` statement or a call to the built-in `eval()` function. This function provides the interface to the compiler, passing the internal parse tree from `ast` to the parser, using the source file name specified by the `filename` parameter. The default value supplied for `filename` indicates that the source was an AST object.

Compiling an AST object may result in exceptions related to compilation; an example would be a `SyntaxError` caused by the parse tree for `del f(0)`; this statement is considered legal within the formal grammar for Python but is not a legal language construct. The `SyntaxError` raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the `parser` module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

### 17.1.3 Queries on AST Objects

Two functions are provided which allow an application to determine if an AST was created as an expression or a suite. Neither of these functions can be used to determine if an AST was created from source code via `expr()` or `suite()` or from a parse tree via `sequence2ast()`.

**isexpr** (*ast*)

When `ast` represents an `eval` form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compileast()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

**issuite** (*ast*)

This function mirrors `isexpr()` in that it reports whether an AST object represents an `exec` form, com-
monly known as a “suite.” It is not safe to assume that this function is equivalent to ‘not isexpr(ast), as additional syntactic fragments may be supported in the future.

17.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception ParserError

Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built in SyntaxError thrown during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to sequence2ast() and an explanatory string. Calls to sequence2ast() need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions compileast(), expr(), and suite() may throw exceptions which are normally thrown by the parsing and compilation process. These include the built in exceptions MemoryError, OverflowError, SyntaxError, and SystemError. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

17.1.5 AST Objects

Ordered and equality comparisons are supported between AST objects. Pickling of AST objects (using the pickle module) is also supported.

ASTType

The type of the objects returned by expr(), suite() and sequence2ast().

AST objects have the following methods:

compile([filename])

Same as compileast(ast, filename).

isexpr()

Same as isexpr(ast).

issuite()

Same as issuite(ast).

tolist([line_info])

Same as ast2list(ast, line_info).

totuple([line_info])

Same as ast2tuple(ast, line_info).

17.1.6 Examples

The parser modules allows operations to be performed on the parse tree of Python source code before the bytecode is generated, and provides for inspection of the parse tree for information gathering purposes. Two examples are presented. The simple example demonstrates emulation of the compile() built-in function and the complex example shows the use of a parse tree for information discovery.
Emulation of compile()

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the parser module to produce an intermediate data structure is equivalent to the code

```python
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the parser module is somewhat longer, and allows the intermediate internal parse tree to be retained as an AST object:

```python
>>> import parser
>>> ast = parser.expr('a + 5')
>>> code = ast.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both AST and code objects can package this code into readily available functions:

```python
import parser

def load_suite(source_string):
    ast = parser.suite(source_string)
    return ast, ast.compile()

def load_expression(source_string):
    ast = parser.expr(source_string)
    return ast, ast.compile()
```

Information Discovery

Some applications benefit from direct access to the parse tree. The remainder of this section demonstrates how the parse tree provides access to module documentation defined in docstrings without requiring that the code being examined be loaded into a running interpreter via import. This can be very useful for performing analyses of untrusted code.

Generally, the example will demonstrate how the parse tree may be traversed to distill interesting information. Two functions and a set of classes are developed which provide programmatic access to high level function and class definitions provided by a module. The classes extract information from the parse tree and provide access to the information at a useful semantic level, one function provides a simple low-level pattern matching capability, and the other function defines a high-level interface to the classes by handling file operations on behalf of the caller. All source files mentioned here which are not part of the Python installation are located in the ‘Demo/parser/’ directory of the distribution.

The dynamic nature of Python allows the programmer a great deal of flexibility, but most modules need only a limited measure of this when defining classes, functions, and methods. In this example, the only definitions that will be considered are those which are defined in the top level of their context, e.g., a function defined by a def statement at
column zero of a module, but not a function defined within a branch of an \texttt{if}...\texttt{else} construct, though there are some good reasons for doing so in some situations. Nesting of definitions will be handled by the code developed in the example.

To construct the upper-level extraction methods, we need to know what the parse tree structure looks like and how much of it we actually need to be concerned about. Python uses a moderately deep parse tree so there are a large number of intermediate nodes. It is important to read and understand the formal grammar used by Python. This is specified in the file ‘Grammar/Grammar’ in the distribution. Consider the simplest case of interest when searching for docstrings: a module consisting of a docstring and nothing else. (See file ‘docstring.py’.)


Using the interpreter to take a look at the parse tree, we find a bewildering mass of numbers and parentheses, with the documentation buried deep in nested tuples.

```python
>>> import parser
>>> import pprint
>>> ast = parser.suite(open('docstring.py').read())
>>> tup = ast.totuple()
>>> pprint.pprint(tup)
(257,
 (264,
  (265,
   (266,
    (267,
     (307,
      (287,
       (288,
        (289,
         (290,
          (292,
           (293,
            (294,
             (295,
              (296,
               (297,
                (298,
                 (299,
                  (300, (3,"""Some documentation.""""))))))))))))),
  (4,''))),
 (4,''),
 (0,''))
```

The numbers at the first element of each node in the tree are the node types; they map directly to terminal and non-terminal symbols in the grammar. Unfortunately, they are represented as integers in the internal representation, and the Python structures generated do not change that. However, the \texttt{symbol} and \texttt{token} modules provide symbolic names for the node types and dictionaries which map from the integers to the symbolic names for the node types.

In the output presented above, the outermost tuple contains four elements: the integer 257 and three additional tuples. Node type 257 has the symbolic name \texttt{file_input}. Each of these inner tuples contains an integer as the first element; these integers, 264, 4, and 0, represent the node types \texttt{stmt}, \texttt{NEWLINE}, and \texttt{ENDMARKER}, respectively. Note that these values may change depending on the version of Python you are using; consult ‘symbol.py’ and ‘token.py’ for details of the mapping. It should be fairly clear that the outermost node is related primarily to the input source rather than the contents of the file, and may be disregarded for the moment. The \texttt{stmt} node is much more interesting. In
particular, all docstrings are found in subtrees which are formed exactly as this node is formed, with the only difference being the string itself. The association between the docstring in a similar tree and the defined entity (class, function, or module) which it describes is given by the position of the docstring subtree within the tree defining the described structure.

By replacing the actual docstring with something to signify a variable component of the tree, we allow a simple pattern matching approach to check any given subtree for equivalence to the general pattern for docstrings. Since the example demonstrates information extraction, we can safely require that the tree be in tuple form rather than list form, allowing a simple variable representation to be ['variable_name']. A simple recursive function can implement the pattern matching, returning a boolean and a dictionary of variable name to value mappings. (See file 'example.py'.)

```python
from types import ListType, TupleType

def match(pattern, data, vars=None):
    if vars is None:
        vars = {}
    if type(pattern) is ListType:
        vars[pattern[0]] = data
        return 1, vars
    if type(pattern) is not TupleType:
        return (pattern == data), vars
    if len(data) != len(pattern):
        return 0, vars
    for pattern, data in map(None, pattern, data):
        same, vars = match(pattern, data, vars)
        if not same:
            break
    return same, vars
```

Using this simple representation for syntactic variables and the symbolic node types, the pattern for the candidate docstring subtrees becomes fairly readable. (See file 'example.py'.)
import symbol
import token
DOCSTRING_STMT_PATTERN = (
symbol.stmt,
(symbol.simple_stmt,
(symbol.small_stmt,
(symbol.expr_stmt,
(symbol.testlist,
(symbol.test,
(symbol.and_test,
(symbol.not_test,
(symbol.comparison,
(symbol.expr,
(symbol.xor_expr,
(symbol.and_expr,
(symbol.shift_expr,
(symbol.arith_expr,
(symbol.term,
(symbol.factor,
(symbol.power,
(symbol.atom,
(token.STRING, [’docstring’])
)))))))))))))))),
(token.NEWLINE, ’’)
))

Using the match() function with this pattern, extracting the module docstring from the parse tree created previously
is easy:
>>> found, vars = match(DOCSTRING_STMT_PATTERN, tup[1])
>>> found
1
>>> vars
{’docstring’: ’"""Some documentation.\n"""’}

Once specific data can be extracted from a location where it is expected, the question of where information can be
expected needs to be answered. When dealing with docstrings, the answer is fairly simple: the docstring is the first
stmt node in a code block (file input or suite node types). A module consists of a single file input node,
and class and function definitions each contain exactly one suite node. Classes and functions are readily identified
as subtrees of code block nodes which start with (stmt, (compound stmt, (classdef, ... or (stmt,
(compound stmt, (funcdef, .... Note that these subtrees cannot be matched by match() since it does
not support multiple sibling nodes to match without regard to number. A more elaborate matching function could be
used to overcome this limitation, but this is sufficient for the example.
Given the ability to determine whether a statement might be a docstring and extract the actual string from the statement,
some work needs to be performed to walk the parse tree for an entire module and extract information about the names
defined in each context of the module and associate any docstrings with the names. The code to perform this work is
not complicated, but bears some explanation.
The public interface to the classes is straightforward and should probably be somewhat more flexible. Each “major”
block of the module is described by an object providing several methods for inquiry and a constructor which accepts
at least the subtree of the complete parse tree which it represents. The ModuleInfo constructor accepts an optional
name parameter since it cannot otherwise determine the name of the module.
The public classes include ClassInfo, FunctionInfo, and ModuleInfo.
424

All objects provide the

Chapter 17. Python Language Services


methods `get_name()`, `get_docstring()`, `get_class_names()`, and `get_class_info()`. The `ClassInfo` objects support `get_method_names()` and `get_method_info()` while the other classes provide `get_function_names()` and `get_function_info()`.

Within each of the forms of code block that the public classes represent, most of the required information is in the same form and is accessed in the same way, with classes having the distinction that functions defined at the top level are referred to as “methods.” Since the difference in nomenclature reflects a real semantic distinction from functions defined outside of a class, the implementation needs to maintain the distinction. Hence, most of the functionality of the public classes can be implemented in a common base class, `SuiteInfoBase`, with the accessors for function and method information provided elsewhere. Note that there is only one class which represents function and method information; this parallels the use of the `def` statement to define both types of elements.

Most of the accessor functions are declared in `SuiteInfoBase` and do not need to be overridden by subclasses. More importantly, the extraction of most information from a parse tree is handled through a method called by the `SuiteInfoBase` constructor. The example code for most of the classes is clear when read alongside the formal grammar, but the method which recursively creates new information objects requires further examination. Here is the relevant part of the `SuiteInfoBase` definition from `example.py`:

```python
class SuiteInfoBase:
    _docstring = ''
    _name = ''

def __init__(self, tree = None):
    self._class_info = {}
    self._function_info = {}
    if tree:
        self._extract_info(tree)

def _extract_info(self, tree):
    # extract docstring
    if len(tree) == 2:
        found, vars = match(DOCSTRING_STMT_PATTERN[1], tree[1])
    else:
        found, vars = match(DOCSTRING_STMT_PATTERN, tree[3])
    if found:
        self._docstring = eval(vars['docstring'])
    # discover inner definitions
    for node in tree[1:]:
        if found:
            cstmt = vars['compound']
            if cstmt[0] == symbol.funcdef:
                name = cstmt[2][1]
                self._function_info[name] = FunctionInfo(cstmt)
            elif cstmt[0] == symbol.classdef:
                name = cstmt[2][1]
                self._class_info[name] = ClassInfo(cstmt)
```

After initializing some internal state, the constructor calls the `_extract_info()` method. This method performs the bulk of the information extraction which takes place in the entire example. The extraction has two distinct phases: the location of the docstring for the parse tree passed in, and the discovery of additional definitions within the code block represented by the parse tree.

The initial `if` test determines whether the nested suite is of the “short form” or the “long form.” The short form is used when the code block is on the same line as the definition of the code block, as in
def square(x): "Square an argument."; return x ** 2

while the long form uses an indented block and allows nested definitions:

def make_power(exp):
    "Make a function that raises an argument to the exponent 'exp'."
    def raiser(x, y=exp):
        return x ** y
    return raiser

When the short form is used, the code block may contain a docstring as the first, and possibly only, small_stmt element. The extraction of such a docstring is slightly different and requires only a portion of the complete pattern used in the more common case. As implemented, the docstring will only be found if there is only one small_stmt node in the simple_stmt node. Since most functions and methods which use the short form do not provide a docstring, this may be considered sufficient. The extraction of the docstring proceeds using the match() function as described above, and the value of the docstring is stored as an attribute of the SuiteInfoBase object.

After docstring extraction, a simple definition discovery algorithm operates on the stmt nodes of the suite node. The special case of the short form is not tested; since there are no stmt nodes in the short form, the algorithm will silently skip the single simple_stmt node and correctly not discover any nested definitions.

Each statement in the code block is categorized as a class definition, function or method definition, or something else. For the definition statements, the name of the element defined is extracted and a representation object appropriate to the definition is created with the defining subtree passed as an argument to the constructor. The representation objects are stored in instance variables and may be retrieved by name using the appropriate accessor methods.

The public classes provide any accessors required which are more specific than those provided by the SuiteInfoBase class, but the real extraction algorithm remains common to all forms of code blocks. A high-level function can be used to extract the complete set of information from a source file. (See file ‘example.py’.)

def get_docs(fileName):
    import os
    import parser

    source = open(fileName).read()
    basename = os.path.basename(os.path.splitext(fileName)[0])
    ast = parser.suite(source)
    return ModuleInfo(ast.totuple(), basename)

This provides an easy-to-use interface to the documentation of a module. If information is required which is not extracted by the code of this example, the code may be extended at clearly defined points to provide additional capabilities.

17.2 symbol — Constants used with Python parse trees

This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file ‘Grammar/Grammar’ in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:
sym_name
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing
more human-readable representation of parse trees to be generated.

See Also:
Module parser (section 17.1):
The second example for the parser module shows how to use the symbol module.

17.3 token — Constants used with Python parse trees

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens).
Refer to the file ‘Grammar/Grammar’ in the Python distribution for the definitions of the names in the context of the
language grammar. The specific numeric values which the names map to may change between Python versions.
This module also provides one data object and some functions. The functions mirror definitions in the Python C header
files.

tok_name
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing
more human-readable representation of parse trees to be generated.

ISTERMINAL (x)
Return true for terminal token values.

ISNONTERMINAL (x)
Return true for non-terminal token values.

ISEOF (x)
Return true if x is the marker indicating the end of input.

See Also:
Module parser (section 17.1):
The second example for the parser module shows how to use the symbol module.

17.4 keyword — Testing for Python keywords

This module allows a Python program to determine if a string is a keyword. A single function is provided:

iskeyword (s)
Return true if s is a Python keyword.

17.5 tokenize — Tokenizer for Python source

The tokenize module provides a lexical scanner for Python source code, implemented in Python. The scanner
in this module returns comments as tokens as well, making it useful for implementing “pretty-printers,” including
colorizers for on-screen displays.
The scanner is exposed by a single function:

tokenize (readline[, token eater ])
The tokenize() function accepts two parameters: one representing the input stream, and one providing an
output mechanism for tokenize().
The first parameter, readline, must be a callable object which provides the same interface as the readline()
method of built-in file objects (see section 2.1.7). Each call to the function should return one line of input as a
string.

The second parameter, `tokeneater`, must also be a callable object. It is called with five parameters: the token
type, the token string, a tuple `(srow, scol)` specifying the row and column where the token begins in the
source, a tuple `(erow, ecol)` giving the ending position of the token, and the line on which the token was
found. The line passed is the `logical` line; continuation lines are included.

All constants from the `token` module are also exported from `tokenize`, as are two additional token type values that
might be passed to the `tokeneater` function by `tokenize()`:

- **COMMENT**
  Token value used to indicate a comment.

- **NL**
  Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical
  line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical
  lines.

### 17.6 `tabnanny` — Detection of ambiguous indentation

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and
use the function `check()` described below.

**Warning:** The API provided by this module is likely to change in future releases; such changes may not be backward
compatible.

```python
check(file_or_dir)
```

If `file_or_dir` is a directory and not a symbolic link, then recursively descend the directory tree named by
`file_or_dir`, checking all `.py` files along the way. If `file_or_dir` is an ordinary Python source file, it is checked
for whitespace related problems. The diagnostic messages are written to standard output using the `print` state-
ment.

- **verbose**
  Flag indicating whether to print verbose messages. This is set to true by the `-v` option if called as a script.

- **filename_only**
  Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set
to true by the `-q` option if called as a script.

- **exception NannyNag**
  Raised by `tokeneater()` if detecting an ambiguous indent. Captured and handled in `check()`.

```python
tokeneater(type, token, start, end, line)
```

This function is used by `check()` as a callback parameter to the function `tokenize.tokenize()`.

See Also:

- **Module `tokenize`** (section 17.5): Lexical scanner for Python source code.

### 17.7 `pyclbr` — Python class browser support

The `pyclbr` can be used to determine some limited information about the classes and methods defined in a module.
The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted
from the source code rather than from an imported module, so this module is safe to use with untrusted source code.
This restriction makes it impossible to use this module with modules not implemented in Python, including many
standard and optional extension modules.

**readmodule**(module[, path])

Read a module and return a dictionary mapping class names to class descriptor objects. The parameter `module` should be the name of a module as a string; it may be the name of a module within a package. The `path` parameter should be a sequence, and is used to augment the value of `sys.path`, which is used to locate module source code.

### 17.7.1 Class Descriptor Objects

The class descriptor objects used as values in the dictionary returned by `readmodule()` provide the following data members:

- **module**
  The name of the module defining the class described by the class descriptor.

- **name**
  The name of the class.

- **super**
  A list of class descriptors which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by `readmodule()` are listed as a string with the class name instead of class descriptors.

- **methods**
  A dictionary mapping method names to line numbers.

- **file**
  Name of the file containing the class statement defining the class.

- **lineno**
  The line number of the class statement within the file named by `file`.

### 17.8 `py_compile` — Compile Python source files

The `py_compile` module provides a single function to generate a byte-code file from a source file.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

**compile**(file[, cfile[, dfile]])

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file name `file`. The byte-code is written to `cfile`, which defaults to `file + c` (`o` if optimization is enabled in the current interpreter). If `dfile` is specified, it is used as the name of the source file in error messages instead of `file`.

See Also:

- **Module compileall** (section 17.9):
  Utilities to compile all Python source files in a directory tree.

### 17.9 `compileall` — Byte-compile Python libraries

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree, allowing users without permission to write to the libraries to take advantage of cached byte-code files.
The source file for this module may also be used as a script to compile Python sources in directories named on the
command line or in sys.path.

```python
code_dir(path[, maxlevels[, ddir[, force]]]]
```

Recursively descend the directory tree named by `dir`, compiling all `.py` files along the way. The `maxlevels`
parameter is used to limit the depth of the recursion; it defaults to 10. If `ddir` is given, it is used as the base path
from which the filenames used in error messages will be generated. If `force` is true, modules are re-compiled
even if the timestamps are up to date.

```python
code_path([skip_curdir[, maxlevels[, force]]]]
```

Byte-compile all the `.py` files found along `sys.path`. If `skip_curdir` is true (the default), the current di-
rectory is not included in the search. The `maxlevels` and `force` parameters default to 0 and are passed to the
code_dir() function.

See Also:

Module `py_compile` (section 17.8):

Byte-compile a single source file.

17.10 `dis` — Disassembler for Python byte code

The `dis` module supports the analysis of Python byte code by disassembling it. Since there is no Python assembler,
this module defines the Python assembly language. The Python byte code which this module takes as an input is
defined in the file ‘Include/opcode.h’ and used by the compiler and the interpreter.

Example: Given the function `myfunc`:

```python
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of `myfunc()`:

```python
>>> dis.dis(myfunc)
0    SET_LINENO    1
  3    SET_LINENO    2
  6    LOAD_GLOBAL 0 (len)
  9    LOAD_FAST    0 (alist)
 12   CALL_FUNCTION 1
 15   RETURN_VALUE
 16   LOAD_CONST    0 (None)
 19   RETURN_VALUE
```

The `dis` module defines the following functions and constants:

```python
dis([bytesource])
```

Disassemble the `bytesource` object. `bytesource` can denote either a class, a method, a function, or a code object.
For a class, it disassembles all methods. For a single code sequence, it prints one line per byte code instruction.
If no object is provided, it disassembles the last traceback.

```python
distb([tb])
```

Disassembles the top-of-stack function of a traceback, using the last traceback if none was passed. The instruc-
tion causing the exception is indicated.

```python
disassemble(code[, last])
```

430 Chapter 17. Python Language Services
Disassembles a code object, indicating the last instruction if lasti was provided. The output is divided in the following columns:

1. the current instruction, indicated as ‘--->’,
2. a labelled instruction, indicated with ‘>>’,
3. the address of the instruction,
4. the operation code name,
5. operation parameters, and
6. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

disco(code, lasti)
A synonym for disassemble. It is more convenient to type, and kept for compatibility with earlier Python releases.

opname
Sequence of operation names, indexable using the byte code.

cmp_op
Sequence of all compare operation names.

hasconst
Sequence of byte codes that have a constant parameter.

hasname
Sequence of byte codes that access an attribute by name.

hasjrel
Sequence of byte codes that have a relative jump target.

hasjabs
Sequence of byte codes that have an absolute jump target.

haslocal
Sequence of byte codes that access a local variable.

hascompare
Sequence of byte codes of boolean operations.

17.10.1 Python Byte Code Instructions

The Python compiler currently generates the following byte code instructions.

STOP_CODE
Indicates end-of-code to the compiler, not used by the interpreter.

POP_TOP
Removes the top-of-stack (TOS) item.

ROT_TWO
Swaps the two top-most stack items.

ROT_THREE
Lifts second and third stack item one position up, moves top down to position three.

ROT_FOUR
Lifts second, third and forth stack item one position up, moves top down to position four.
DUP_TOP
Duplicates the reference on top of the stack.

Unary Operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE
Implements \( \text{TOS} = +\text{TOS} \).

UNARY_NEGATIVE
Implements \( \text{TOS} = -\text{TOS} \).

UNARY_NOT
Implements \( \text{TOS} = \text{not} \ \text{TOS} \).

UNARY_CONVERT
Implements \( \text{TOS} = \text{\textquotesingle}\text{TOS}\text{\textquotesingle} \).

UNARY_INVERT
Implements \( \text{TOS} = \bar{\text{TOS}} \).

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER
Implements \( \text{TOS} = \text{TOS1} \ \text{**} \ \text{TOS} \).

BINARY_MULTIPLY
Implements \( \text{TOS} = \text{TOS1} \ \text{*} \ \text{TOS} \).

BINARY_DIVIDE
Implements \( \text{TOS} = \text{TOS1} \ / \ \text{TOS} \).

BINARY_MODULO
Implements \( \text{TOS} = \text{TOS1} \ \% \ \text{TOS} \).

BINARY_ADD
Implements \( \text{TOS} = \text{TOS1} \ + \ \text{TOS} \).

BINARY_SUBTRACT
Implements \( \text{TOS} = \text{TOS1} \ - \ \text{TOS} \).

BINARY_SUBSCR
Implements \( \text{TOS} = \text{TOS1} \[	ext{TOS}\] \).

BINARY_LSHIFT
Implements \( \text{TOS} = \text{TOS1} \ \ll \ \text{TOS} \).

BINARY_RSHIFT
Implements \( \text{TOS} = \text{TOS1} \ \gg \ \text{TOS} \).

BINARY_AND
Implements \( \text{TOS} = \text{TOS1} \ \& \ \text{TOS} \).

BINARY_XOR
Implements \( \text{TOS} = \text{TOS1} \ \text{^} \ \text{TOS} \).

BINARY_OR
Implements \( \text{TOS} = \text{TOS1} \ \| \ \text{TOS} \).

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

INPLACE_POWER
Implements in-place \( \text{TOS} = \text{TOS1} \ \text{**} \ \text{TOS} \).
INPLACE_MULTIPLY
Implements in-place $TOS = TOS1 \times TOS$.

INPLACE_DIVIDE
Implements in-place $TOS = TOS1 \div TOS$.

INPLACE_MODULO
Implements in-place $TOS = TOS1 \mod TOS$.

INPLACE_ADD
Implements in-place $TOS = TOS1 + TOS$.

INPLACE_SUBTRACT
Implements in-place $TOS = TOS1 - TOS$.

INPLACE_LSHIFT
Implements in-place $TOS = TOS1 \ll TOS$.

INPLACE_RSHIFT
Implements in-place $TOS = TOS1 \gg TOS$.

INPLACE_AND
Implements in-place $TOS = TOS1 \& TOS$.

INPLACE_XOR
Implements in-place $TOS = TOS1 \^ TOS$.

INPLACE_OR
Implements in-place $TOS = TOS1 | TOS$.

The slice opcodes take up to three parameters.

SLICE+0
Implements $TOS = TOS[:]$.

SLICE+1
Implements $TOS = TOS1[TOS:]$.

SLICE+2
Implements $TOS = TOS1[:TOS1]$.

SLICE+3
Implements $TOS = TOS2[TOS1:TOS]$.

Slice assignment needs even an additional parameter. As any statement, they put nothing on the stack.

STORE_SLICE+0
Implements $TOS[:] = TOS1$.

STORE_SLICE+1
Implements $TOS1[TOS:] = TOS2$.

STORE_SLICE+2
Implements $TOS1[:TOS] = TOS2$.

STORE_SLICE+3
Implements $TOS2[TOS1:TOS] = TOS3$.

DELETE_SLICE+0
Implements del $TOS[:]$.

DELETE_SLICE+1
Implements del $TOS1[TOS:]$.

DELETE_SLICE+2
Implements `del TOS1[:TOS]`.

**DELETE_SLICE+3**

Implements `del TOS2[TOS1:TOS]`.

**STORE_SUBSCR**

Implements `TOS1[TOS] = TOS2`.

**DELETE_SUBSCR**

Implements `del TOS1[TOS]`.

**PRINT_EXPR**

Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with `POP_STACK`.

**PRINT_ITEM**

Prints TOS to the file-like object bound to `sys.stdout`. There is one such instruction for each item in the `print` statement.

**PRINT_ITEM_TO**

Like **PRINT_ITEM**, but prints the item second from TOS to the file-like object at TOS. This is used by the extended print statement.

**PRINT_NEWLINE**

Prints a new line on `sys.stdout`. This is generated as the last operation of a `print` statement, unless the statement ends with a comma.

**PRINT_NEWLINE_TO**

Like **PRINT_NEWLINE**, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.

**BREAK_LOOP**

Terminates a loop due to a `break` statement.

**LOAD_LOCALS**

Pushes a reference to the locals of the current scope on the stack. This is used in the code for a class definition: After the class body is evaluated, the locals are passed to the class definition.

**RETURN_VALUE**

Returns with TOS to the caller of the function.

**IMPORT_STAR**

Loads all symbols not starting with `.` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

**EXEC_STMT**

Implements `exec TOS2, TOS1, TOS`. The compiler fills missing optional parameters with `None`.

**POP_BLOCK**

Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

**END_FINALLY**

Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

**BUILD_CLASS**

Creates a new class object. TOS is the methods dictionary, TOS1 the tuple of the names of the base classes, and TOS2 the class name.

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

**STORE_NAME**  `namei`
Implements name = TOS. namei is the index of name in the attribute co_names of the code object. The compiler tries to use STORE_LOCAL or STORE_GLOBAL if possible.

DELETE_NAME namei
Implements del name, where namei is the index into co_names attribute of the code object.

UNPACK_SEQUENCE count
Unpacks TOS into count individual values, which are put onto the stack right-to-left.

DUP_TOPX count
Duplicate count items, keeping them in the same order. Due to implementation limits, count should be between 1 and 5 inclusive.

STORE_ATTR namei
Implements TOS.name = TOS1, where namei is the index of name in co_names.

DELETE_ATTR namei
Implements del TOS.name, using namei as index into co_names.

STORE_GLOBAL namei
Works as STORE_NAME, but stores the name as a global.

DELETE_GLOBAL namei
Works as DELETE_NAME, but deletes a global name.

LOAD_CONSTANT consti
Pushes ‘co_consts[consti]’ onto the stack.

LOAD_NAME namei
Pushes the value associated with ‘co_names[namei]’ onto the stack.

BUILD_TUPLE count
Creates a tuple consuming count items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST count
Works as BUILD_TUPLE, but creates a list.

BUILD_MAP zero
Pushes a new empty dictionary object onto the stack. The argument is ignored and set to zero by the compiler.

LOAD_ATTR namei
Replaces TOS with getattr(TOS, co_names[namei]).

COMPARE_OP opname
Performs a boolean operation. The operation name can be found in cmp_op[opname].

IMPORT namei
Imports the module co_names[namei]. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent STORE_FAST instruction modifies the namespace.

IMPORT_FROM namei
Loads the attribute co_names[namei] from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a STORE_FAST instruction.

JUMP_FORWARD delta
Increments byte code counter by delta.

JUMP_IF_TRUE delta
If TOS is true, increment the byte code counter by delta. TOS is left on the stack.

JUMP_IF_FALSE delta
If TOS is false, increment the byte code counter by delta. TOS is not changed.

JUMP_ABSOLUTE target
Set byte code counter to target.

**FOR_LOOP**  \(\text{delta}\)
Iterate over a sequence. TOS is the current index, TOS1 the sequence. First, the next element is computed. If the sequence is exhausted, increment byte code counter by \(\text{delta}\). Otherwise, push the sequence, the incremented counter, and the current item onto the stack.

**LOAD_GLOBAL**  \(\text{namei}\)
Loads the global named \(\text{co_names}[\text{namei}]\) onto the stack.

**SETUP_LOOP**  \(\text{delta}\)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of \(\text{delta}\) bytes.

**SETUP_EXCEPT**  \(\text{delta}\)
Pushes a try block from a try-except clause onto the block stack. \(\text{delta}\) points to the first except block.

**SETUP_FINALLY**  \(\text{delta}\)
Pushes a try block from a try-except clause onto the block stack. \(\text{delta}\) points to the finally block.

**LOAD_FAST**  \(\text{var_num}\)
Pushes a reference to the local \(\text{co_varnames}[\text{var_num}]\) onto the stack.

**STORE_FAST**  \(\text{var_num}\)
 Stores TOS into the local \(\text{co_varnames}[\text{var_num}]\).

**DELETE_FAST**  \(\text{var_num}\)
Deletes local \(\text{co_varnames}[\text{var_num}]\).

**LOAD_CLOSURE**  \(i\)
Pushes a reference to the cell contained in slot \(i\) of the cell and free variable storage. The name of the variable is \(\text{co_cellvars}[i]\) if \(i\) is less than the length of \(\text{co_cellvars}\). Otherwise it is \(\text{co_freevars}[i - \text{len(\text{co_cellvars})}]\).

**LOAD_DEREF**  \(i\)
Loads the cell contained in slot \(i\) of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

**STORE_DEREF**  \(i\)
Stores TOS into the cell contained in slot \(i\) of the cell and free variable storage.

**SET_LINENO**  \(\text{lineno}\)
Sets the current line number to \(\text{lineno}\).

**RAISE_VARARGS**  \(\text{argc}\)
Raises an exception. \(\text{argc}\) indicates the number of parameters to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

**CALL_FUNCTION**  \(\text{argc}\)
Calls a function. The low byte of \(\text{argc}\) indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack.

**MAKE_FUNCTION**  \(\text{argc}\)
Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have \(\text{argc}\) default parameters, which are found below TOS.

**MAKE_CLOSURE**  \(\text{argc}\)
Creates a new function object, sets its \(\text{func_closure}\) slot, and pushes it on the stack. TOS is the code associated with the function. If the code object has \(N\) free variables, the next \(N\) items on the stack are the cells for these variables. The function also has \(\text{argc}\) default parameters, where are found before the cells.
**BUILD_SLICE**  *argc*

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the `slice()` built-in function for more information.

**EXTENDED_ARG**  *ext*

Prefixes any opcode which has an argument too big to fit into the default two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode’s argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

**CALL_FUNCTION_VAR**  *argc*

Calls a function. *argc* is interpreted as in `CALL_FUNCTION`. The top element on the stack contains the variable argument list, followed by keyword and positional arguments.

**CALL_FUNCTION_KW**  *argc*

Calls a function. *argc* is interpreted as in `CALL_FUNCTION`. The top element on the stack contains the keyword arguments dictionary, followed by explicit keyword and positional arguments.

**CALL_FUNCTION_VAR_KW**  *argc*

Calls a function. *argc* is interpreted as in `CALL_FUNCTION`. The top element on the stack contains the keyword arguments dictionary, followed by the variable-arguments tuple, followed by explicit keyword and positional arguments.
SGI IRIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to SGI’s IRIX operating system (versions 4 and 5).

- **al**: Audio functions on the SGI.
- **AL**: Constants used with the al module.
- **cd**: Interface to the CD-ROM on Silicon Graphics systems.
- **fl**: FORMS library interface for GUI applications.
- **FL**: Constants used with the fl module.
- **flp**: Functions for loading stored FORMS designs.
- **fm**: Font Manager interface for SGI workstations.
- **gl**: Functions from the Silicon Graphics Graphics Library.
- **DEVICE**: Constants used with the gl module.
- **GL**: Constants used with the gl module.
- **imgfile**: Support for SGI imglib files.
- **jpeg**: Read and write image files in compressed JPEG format.

### 18.1 al — Audio functions on the SGI

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You’ll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with ‘AL’ prefixed to their name. Symbolic constants from the C header file `<audio.h>` are defined in the standard module `AL`, see below.

**Warning**: the current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size — there is no documented upper limit.)

The module defines the following functions:

- **openport**(name, direction[, config])
  
  The name and direction arguments are strings. The optional config argument is a configuration object as returned by `newconfig()`. The return value is an `audio port object`; methods of audio port objects are described below.

- **newconfig**()
  
  The return value is a new `audio configuration object`; methods of audio configuration objects are described below.

- **queryparams**(device)
The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

\texttt{getparams} (device, list)

The \textit{device} argument is an integer. The list argument is a list such as returned by queryparams(); it is modified in place (!).

\texttt{setparams} (device, list)

The \textit{device} argument is an integer. The \textit{list} argument is a list such as returned by queryparams().

18.1.1 Configuration Objects

Configuration objects (returned by \texttt{newconfig()}) have the following methods:

\texttt{getqueuesize} ()

Return the queue size.

\texttt{setqueuesize} (size)

Set the queue size.

\texttt{getwidth} ()

Get the sample width.

\texttt{setwidth} (width)

Set the sample width.

\texttt{getchannels} ()

Get the channel count.

\texttt{setchannels} (nchannels)

Set the channel count.

\texttt{getsampfmt} ()

Get the sample format.

\texttt{setsampfmt} (sampfmt)

Set the sample format.

\texttt{getfloatmax} ()

Get the maximum value for floating sample formats.

\texttt{setfloatmax} (floatmax)

Set the maximum value for floating sample formats.

18.1.2 Port Objects

Port objects, as returned by \texttt{openport()}, have the following methods:

\texttt{closeport} ()

Close the port.

\texttt{getfd} ()

Return the file descriptor as an int.

\texttt{getfilled} ()

Return the number of filled samples.

\texttt{getfillable} ()

Return the number of fillable samples.

\texttt{readsamps} (nsamples)
Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).

writesamps(samples)
Write samples into the queue, blocking if necessary. The samples are encoded as described for the readsamps() return value.

getfillpoint()
Return the ‘fill point’.

setfillpoint(fillpoint)
Set the ‘fill point’.

getconfig()
Return a configuration object containing the current configuration of the port.

setconfig(config)
Set the configuration from the argument, a configuration object.

getstatus(list)
Get status information on last error.

18.2 AL — Constants used with the al module

This module defines symbolic constants needed to use the built-in module al (see above); they are equivalent to those defined in the C header file <audio.h> except that the name prefix ‘AL’ is omitted. Read the module source for a complete list of the defined names. Suggested use:

```python
import al
from AL import *
```

18.3 cd — CD-ROM access on SGI systems

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with `open()` and creates a parser to parse the data from the CD with `createparser()`. The object returned by `open()` can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into tracks or programs (the terms are used interchangeably). Tracks can be subdivided into indices. An audio CD contains a table of contents which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module `cd` defines the following functions and constants:

createparser()
Create and return an opaque parser object. The methods of the parser object are described below.
**msftoframe** *(minutes, seconds, frames)*

Converts a *(minutes, seconds, frames)* triple representing time in absolute time code into the corresponding CD frame number.

**open** *

Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. `/dev/scsi/sc0d4l0`, or *None*. If omitted or *None*, the hardware inventory is consulted to locate a CD-ROM drive. The *mode*, if not omitted, should be the string `'r'`.

The module defines the following variables:

**exception error**

Exception raised on various errors.

**DATASIZE**

The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type *audio*.

**BLOCKSIZE**

The size of one uninterpreted frame of audio data.

The following variables are states as returned by *getstatus()*:

**READY**

The drive is ready for operation loaded with an audio CD.

**NODISC**

The drive does not have a CD loaded.

**CDROM**

The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

**ERROR**

An error occurred while trying to read the disc or its table of contents.

**PLAYING**

The drive is in CD player mode playing an audio CD through its audio jacks.

**PAUSED**

The drive is in CD layer mode with play paused.

**STILL**

The equivalent of **PAUSED** on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

*audio*

*pnum*

*index*

*ptime*

*atime*

*catalog*

*ident*

*control*

Integer constants describing the various types of parser callbacks that can be set by the *addcallback()* method of CD parser objects (see below).

### 18.3.1 Player Objects

Player objects (returned by *open()* ) have the following methods:
allowremoval ()
Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

bestreadsize ()
Returns the best value to use for the num_frames parameter of the readda () method. Best is defined as the
value that permits a continuous flow of data from the CD-ROM drive.

close ()
Frees the resources associated with the player object. After calling close (), the methods of the object should
no longer be used.

eject ()
Ejects the caddy from the CD-ROM drive.

getstatus ()
Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple
with the following values: state, track, rtime, atime, ttime, first, last, scsi_audio, cur_block. rtime is the time
relative to the start of the current track; atime is the time relative to the beginning of the disc; ttime is the total
time on the disc. For more information on the meaning of the values, see the man page CDgetstatus(3dm). The
value of state is one of the following: ERROR, NODISC, READY, PLAYING, PAUSED, STILL, or CDROM.

gettrackinfo (track)
Returns information about the specified track. The returned information is a tuple consisting of two elements,
the start time of the track and the duration of the track.

msftoblock (min, sec, frame)
Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logi-
cal block number for the given CD-ROM drive. You should use msftoframe () rather than msftoblock ()
for comparing times. The logical block number differs from the frame number by an offset required by certain
CD-ROM drives.

play (start, play)
Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the
CD-ROM drive’s headphone and audio jacks (if fitted). Play stops at the end of the disc. start is the number of
the track at which to start playing the CD; if play is 0, the CD will be set to an initial paused state. The method
togglepause () can then be used to commence play.

playabs (minutes, seconds, frames, play)
Like play (), except that the start is given in minutes, seconds, and frames instead of a track number.

playtrack (start, play)
Like play (), except that playing stops at the end of the track.

playtrackabs (track, minutes, seconds, frames, play)
Like play (), except that playing begins at the specified absolute time and ends at the end of the specified
track.

preventremoval ()
Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

readda (num_frames)
Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a
string representing the audio frames. This string can be passed unaltered to the parseframe () method of the
parser object.

seek (minutes, seconds, frames)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The
pointer is set to an absolute time code location specified in minutes, seconds, and frames. The return value is the
logical block number to which the pointer has been set.

seekblock (block)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

**seektrack**(track)

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

**stop**()

Stops the current playing operation.

**togglepause**()

Pauses the CD if it is playing, and makes it play if it is paused.

### 18.3.2 Parser Objects

Parser objects (returned by `createparser()` have the following methods:

**addcallback**(type, func, arg)

Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the `cd` module level (see above). The callback is called as follows: `func(arg, type, data)`, where `arg` is the user supplied argument, `type` is the particular type of callback, and `data` is the data returned for this `type` of callback. The type of the data depends on the `type` of callback as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio</td>
<td>String which can be passed unmodified to <code>al.writesamps()</code></td>
</tr>
<tr>
<td>pnum</td>
<td>Integer giving the program (track) number.</td>
</tr>
<tr>
<td>index</td>
<td>Integer giving the index number.</td>
</tr>
<tr>
<td>ptime</td>
<td>Tuple consisting of the program time in minutes, seconds, and frames.</td>
</tr>
<tr>
<td>atime</td>
<td>Tuple consisting of the absolute time in minutes, seconds, and frames.</td>
</tr>
<tr>
<td>catalog</td>
<td>String of 13 characters, giving the catalog number of the CD.</td>
</tr>
<tr>
<td>ident</td>
<td>String of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.</td>
</tr>
<tr>
<td>control</td>
<td>Integer giving the control bits from the CD subcode data</td>
</tr>
</tbody>
</table>

**deleteparser**()

Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

**parseframe**(frame)

Parses one or more frames of digital audio data from a CD such as returned by `readda()`. It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then `parseframe()` executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

**removecallback**(type)

Removes the callback for the given `type`.

**resetparser**()

Resets the fields of the parser used for tracking subcodes to an initial state. `resetparser()` should be called after the disc has been changed.
18.4  fl — FORMS library interface for GUI applications

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous ftp from host ‘ftp.cs.ruu.nl’, directory ‘SGI/FORMS’. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial ‘fl_’ from their name. Constants used by the library are defined in module FL described below.

The creation of objects is a little different in Python than in C: instead of the ‘current form’ maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions fl_addto_form() and fl_end_form(), and the equivalent of fl_bgn_form() is called fl_make_form().

Watch out for the somewhat confusing terminology: FORMS uses the word object for the buttons, sliders etc. that you can place in a form. In Python, ‘object’ means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn’t too confusing.

There are no ‘free objects’ in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Please note: importing fl implies a call to the GL function foreground() and to the FORMS routine fl_init().

18.4.1  Functions Defined in Module fl

Module fl defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

make_form(type, width, height)
Create a form with given type, width and height. This returns a form object, whose methods are described below.

do_forms()
The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value FL_EVENT.

check_forms()
Check for FORMS events. Returns what do_forms() above returns, or None if there is no event that immediately needs interaction.

set_event_callback(function)
Set the event callback function.

set_graphics_mode(rgbmode, doublebuffering)
Set the graphics modes.

get_rgbmode()
Return the current rgb mode. This is the value of the C global variable fl_rgbmode.

show_message(str1, str2, str3)
Show a dialog box with a three-line message and an OK button.

show_question(str1, str2, str3)
Show a dialog box with a three-line message and YES and NO buttons. It returns 1 if the user pressed YES, 0 if NO.

show_choice(str1, str2, str3, but1[, but2[, but3 ]])
Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked.
by the user (1, 2 or 3).

`show_input(prompt, default)`

Show a dialog box with a one-line prompt message and text field in which the user can enter a string. The second argument is the default input string. It returns the string value as edited by the user.

`show_file_selector(message, directory, pattern, default)`

Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or None if the user presses Cancel.

`get_directory()`

`get_pattern()`

`get_filename()`

These functions return the directory, pattern and filename (the tail part only) selected by the user in the last `show_file_selector()` call.

`qdevice(dev)`

`unqdevice(dev)`

`isqueued(dev)`

`qtest()`

`qread()`

`qreset()`

`qenter(dev, val)`

`get_mouse()`

`tie(button, valuator1, valuator2)`

These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to handle some GL events yourself when using `fl.do_events()`. When a GL event is detected that FORMS cannot handle, `fl.do_forms()` returns the special value FL.EVENT and you should call `fl.qread()` to read the event from the queue. Don’t use the equivalent GL functions!

`color()`

`mapcolor()`

`getmcolor()`

See the description in the FORMS documentation of `fl_color()`, `fl_mapcolor()` and `fl_getmcolor()`.

18.4.2 Form Objects

Form objects (returned by `make_form()` above) have the following methods. Each method corresponds to a C function whose name is prefixed with ‘fl_’; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the `add_*()` methods return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

`show_form(placement, bordertype, name)`

Show the form.

`hide_form()`

Hide the form.

`redraw_form()`

Redraw the form.

`set_form_position(x, y)`

Set the form’s position.
freeze_form()
    Freeze the form.
unfreeze_form()
    Unfreeze the form.
activate_form()
    Activate the form.
deactivate_form()
    Deactivate the form.
bgn_group()
    Begin a new group of objects; return a group object.
end_group()
    End the current group of objects.
find_first()
    Find the first object in the form.
find_last()
    Find the last object in the form.
add_box(type, x, y, w, h, name)
    Add a box object to the form. No extra methods.
add_text(type, x, y, w, h, name)
    Add a text object to the form. No extra methods.
add_clock(type, x, y, w, h, name)
    Add a clock object to the form.
    Method: get_clock().
add_button(type, x, y, w, h, name)
    Add a button object to the form.
    Methods: get_button(), set_button().
add_lightbutton(type, x, y, w, h, name)
    Add a lightbutton object to the form.
    Methods: get_button(), set_button().
add_roundbutton(type, x, y, w, h, name)
    Add a roundbutton object to the form.
    Methods: get_button(), set_button().
add_slider(type, x, y, w, h, name)
    Add a slider object to the form.
    Methods: get_slider_value(), set_slider_value(), set_slider_bounds(),
              get_slider_bounds(), set_slider_return(), set_slider_size(),
              set_slider_precision(), set_slider_step().
add_valslider(type, x, y, w, h, name)
    Add a valslider object to the form.
    Methods: get_slider_value(), set_slider_value(), set_slider_bounds(),
              get_slider_bounds(), set_slider_return(), set_slider_size(),
              set_slider_precision(), set_slider_step().
add_dial(type, x, y, w, h, name)
    Add a dial object to the form.
    Methods: set_dial_value(), get_dial_value(), set_dial_bounds(),
              get_dial_bounds().
add_positioner(type, x, y, w, h, name)
Add a positioner object to the form.
Methods: set_positioner_xvalue(), set_positioner_yvalue(),
set_positioner_xbounds(), set_positioner_ybounds(), get_positioner_xvalue(),
get_positioner_yvalue(), get_positioner_xbounds(), get_positioner_ybounds().

add_counter(type, x, y, w, h, name)
Add a counter object to the form.
Methods: set_counter_value(), get_counter_value(), set_counter_bounds(),
set_counter_step(), set_counter_precision(), set_counter_return().

add_input(type, x, y, w, h, name)
Add a input object to the form.
Methods: set_input(), get_input(), set_input_color(), set_input_return().

add_menu(type, x, y, w, h, name)
Add a menu object to the form.
Methods: set_menu(), get_menu(), addto_menu().

add_choice(type, x, y, w, h, name)
Add a choice object to the form.
Methods: set_choice(), get_choice(), clear_choice(), addto_choice(),
replace_choice(), delete_choice(), get_choice_text(), set_choice_fontsize(),
set_choice_fontstyle().

add_browser(type, x, y, w, h, name)
Add a browser object to the form.
Methods: set_browser_topline(), clear_browser(), add_browser_line(),
addto_browser(), insert_browser_line(), delete_browser_line(),
replace_browser_line(), get_browser_line(), load_browser(),
get_browser_maxline(), select_browser_line(), deselect_browser_line(),
deselect_browser(), isselected_browser_line(), get_browser(),
set_browser_fontsize(), set_browser_fontstyle(), set_browser_specialkey().

add_timer(type, x, y, w, h, name)
Add a timer object to the form.
Methods: set_timer(), get_timer().

Form objects have the following data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>int (read-only)</td>
<td>GL window id</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>form width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>form height</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>form x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>form y origin</td>
</tr>
<tr>
<td>deactivated</td>
<td>int</td>
<td>nonzero if form is deactivated</td>
</tr>
<tr>
<td>visible</td>
<td>int</td>
<td>nonzero if form is visible</td>
</tr>
<tr>
<td>frozen</td>
<td>int</td>
<td>nonzero if form is frozen</td>
</tr>
<tr>
<td>doublebuf</td>
<td>int</td>
<td>nonzero if double buffering on</td>
</tr>
</tbody>
</table>

18.4.3 FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

set_call_back(function, argument)
Set the object’s callback function and argument. When the object needs interaction, the callback function will be
called with two arguments: the object, and the callback argument. (FORMS objects without a callback function are returned by \texttt{fl.do_forms()} or \texttt{fl.check_forms()} when they need interaction.) Call this method without arguments to remove the callback function.

\texttt{delete_object()}
Delete the object.
\texttt{show_object()}
Show the object.
\texttt{hide_object()}
Hide the object.
\texttt{redraw_object()}
Redraw the object.
\texttt{freeze_object()}
Freeze the object.
\texttt{unfreeze_object()}
Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>objclass</td>
<td>int (read-only)</td>
<td>object class</td>
</tr>
<tr>
<td>type</td>
<td>int (read-only)</td>
<td>object type</td>
</tr>
<tr>
<td>boxtype</td>
<td>int</td>
<td>box type</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>y origin</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>height</td>
</tr>
<tr>
<td>col1</td>
<td>int</td>
<td>primary color</td>
</tr>
<tr>
<td>col2</td>
<td>int</td>
<td>secondary color</td>
</tr>
<tr>
<td>align</td>
<td>int</td>
<td>alignment</td>
</tr>
<tr>
<td>lcol</td>
<td>int</td>
<td>label color</td>
</tr>
<tr>
<td>lsize</td>
<td>float</td>
<td>label font size</td>
</tr>
<tr>
<td>label</td>
<td>string</td>
<td>label string</td>
</tr>
<tr>
<td>lstyle</td>
<td>int</td>
<td>label style</td>
</tr>
<tr>
<td>pushed</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>focus</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>belowmouse</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>frozen</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>active</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>input</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>visible</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>radio</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
<tr>
<td>automatic</td>
<td>int (read-only)</td>
<td>(see FORMS docs)</td>
</tr>
</tbody>
</table>

18.5 \texttt{FL} — Constants used with the \texttt{fl} module

This module defines symbolic constants needed to use the built-in module \texttt{fl} (see above); they are equivalent to those defined in the C header file \texttt{<forms.h>} except that the name prefix ‘FL’ is omitted. Read the module source for a complete list of the defined names. Suggested use:
**18.6 flp — Functions for loading stored FORMS designs**

This module defines functions that can read form definitions created by the ‘form designer’ (fdesign) program that comes with the FORMS library (see module fl above).

For now, see the file ‘flp.doc’ in the Python library source directory for a description.

XXX A complete description should be inserted here!

**18.7 fm — Font Manager interface**

This module provides access to the IRIS Font Manager library. It is available only on Silicon Graphics machines. See also: 4Sight User’s Guide, section 1, chapter 5: “Using the IRIS Font Manager.”

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

- **init()**
  - Initialization function. Calls fminit(). It is normally not necessary to call this function, since it is called automatically the first time the fm module is imported.

- **findfont(fontname)**
  - Return a font handle object. Calls fmfindfont(fontname).

- **enumerate()**
  - Returns a list of available font names. This is an interface to fmenumerate().

- **prstr(string)**
  - Render a string using the current font (see the setfont() font handle method below). Calls fmprstr(string).

- **setpath(string)**
  - Sets the font search path. Calls fmsetpath(string). (XXX Does not work!??)

- **fontpath()**
  - Returns the current font search path.

Font handle objects support the following operations:

- **scalefont(factor)**
  - Returns a handle for a scaled version of this font. Calls fmscalefont(fh, factor).

- **setfont()**
  - Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls fmsetfont(fh).

- **getfontname()**
  - Returns this font’s name. Calls fmgetfontname(fh).

- **getcomment()**
  - Returns the comment string associated with this font. Raises an exception if there is none. Calls
getfontinfo()

Returns a tuple giving some pertinent data about this font. This is an interface to fmgetfontinfo(). The returned tuple contains the following numbers: \((\text{printermatched}, \text{fixed\_width}, \text{xorig}, \text{yorig}, \text{xsize}, \text{ysize}, \text{height}, \text{nglyphs})\).

getstrwidth(string)

Returns the width, in pixels, of \(\text{string}\) when drawn in this font. Calls fmgetstrwidth(fh, string).

18.8 gl — Graphics Library interface

This module provides access to the Silicon Graphics Graphics Library. It is available only on Silicon Graphics machines.

Warning: Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.
- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.
- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.
- All string and character arguments are represented by Python strings, for instance, \(\text{winopen}('\text{Hi There}!'\text{)}\) and \(\text{rotate}(900, 'z')\).
- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call

\[
\text{lmdef}(\text{deftype}, \text{index}, \text{np}, \text{props})
\]

is translated to Python as

\[
\text{lmdef}(\text{deftype}, \text{index}, \text{props})
\]

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call

\[
\text{getmcolor}(i, &\text{red}, &\text{green}, &\text{blue})
\]

is translated to Python as

\[
\text{red}, \text{green}, \text{blue} = \text{getmcolor}(i)
\]
The following functions are non-standard or have special argument conventions:

**varray** (argument)
Equivalent to but faster than a number of *v3d()* calls. The *argument* is a list (or tuple) of points. Each point must be a tuple of coordinates \((x, y, z)\) or \((x, y)\). The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double precision points by assuming \(z = 0.0\) if necessary (as indicated in the man page), and for each point *v3d()* is called.

**narray()**
Equivalent to but faster than a number of *n3f* and *v3f* calls. The argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates \((x, y, z)\). Three coordinates must be given. Float and int values may be mixed. For each pair, *n3f()* is called for the normal, and then *v3f()* is called for the point.

**vnarray()**
Similar to *narray()* but the pairs have the point first and the normal second.

**nurbssurface** (s_k, t_k, ctl, s_ord, t_ord, type)
Defines a nurbs surface. The dimensions of *ctl[]* are computed as follows: \([\text{len}(s_k) - s_{ord}], [\text{len}(t_k) - t_{ord}]\).

**nurbscurve** (knots, ctlpoints, order, type)
Defines a nurbs curve. The length of ctlpoints is \(\text{len}(knots) - \text{order}\).

**pwlcurve** (points, type)
Defines a piecewise-linear curve. *points* is a list of points. *type* must be N_ST.

**pick** (n)
**select** (n)
The only argument to these functions specifies the desired size of the pick or select buffer.

**endpick()**
**endselct()**
These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)

main()

See Also:
An interface to OpenGL is also available; see information about David Ascher’s PyOpenGL online at http://starship.python.net/crew/da/PyOpenGL/. This may be a better option if support for SGI hardware from before about 1996 is not required.

18.9 DEVICE — Constants used with the gl module

This module defines the constants used by the Silicon Graphics Graphics Library that C programmers find in the header file <gl/device.h>. Read the module source file for details.

18.10 GL — Constants used with the gl module

This module contains constants used by the Silicon Graphics Graphics Library from the C header file <gl/gl.h>. Read the module source file for details.

18.11 imgfile — Support for SGI imglib files

The imgfile module allows Python programs to access SGI imglib image files (also known as `.rgb` files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

**exception** error

This exception is raised on all errors, such as unsupported file type, etc.

**getsizes** (file)

This function returns a tuple \((x, y, z)\) where \(x\) and \(y\) are the size of the image in pixels and \(z\) is the number
of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

**read (file)**
This function reads and decodes the image on the specified file, and returns it as a Python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite()`, for instance.

**readscaled (file, x, y, filter[, blur])**
This function is identical to read but it returns an image that is scaled to the given x and y sizes. If the filter and blur parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer-generated images.

Alternatively, you can specify a filter to use to smoothen the image after scaling. The filter forms supported are 'impulse', 'box', 'triangle', 'quadratic' and 'gaussian'. If a filter is specified blur is an optional parameter specifying the blurriness of the filter. It defaults to 1.0.

`readscaled()` makes no attempt to keep the aspect ratio correct, so that is the users’ responsibility.

**ttob (flag)**
This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.

**write (file, data, x, y, z)**
This function writes the RGB or greyscale data in `data` to image file `file`. `x` and `y` give the size of the image, `z` is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by `gl.lrectread()`.

### 18.12 jpeg — Read and write JPEG files

The module `jpeg` provides access to the jpeg compressor and decompressor written by the Independent JPEG Group (IJG). JPEG is a standard for compressing pictures; it is defined in ISO 10918. For details on JPEG or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

A portable interface to JPEG image files is available with the Python Imaging Library (PIL) by Fredrik Lundh. Information on PIL is available at [http://www.pythonware.com/products/pil/](http://www.pythonware.com/products/pil/).

The `jpeg` module defines an exception and some functions.

**exception error**
Exception raised by `compress()` and `decompress()` in case of errors.

**compress (data, w, h, b)**
Treat data as a pixmap of width `w` and height `h`, with `b` bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that `gl.lrectread()` return data can immediately be passed to `compress()`. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. `compress()` returns a string that contains the compressed picture, in JFIF format.

**decompress (data)**
Data is a string containing a picture in JFIF format. It returns a tuple `(data, width, height, bytesperpixel)`. Again, the data is suitable to pass to `gl.lrectwrite()`.

**setoption (name, value)**
Set various options. Subsequent `compress()` and `decompress()` calls will use these options. The following options are available:
<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>'forcegray'</td>
<td>Force output to be grayscale, even if input is RGB.</td>
</tr>
<tr>
<td>'quality'</td>
<td>Set the quality of the compressed image to a value between 0 and 100 (default is 75). This only affects compression.</td>
</tr>
<tr>
<td>'optimize'</td>
<td>Perform Huffman table optimization. Takes longer, but results in smaller compressed image. This only affects compression.</td>
</tr>
<tr>
<td>'smooth'</td>
<td>Perform inter-block smoothing on uncompressed image. Only useful for low-quality images. This only affects decompression.</td>
</tr>
</tbody>
</table>

See Also:

**JPEG Still Image Data Compression Standard**

The canonical reference for the JPEG image format, by Pennebaker and Mitchell.

**Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines**

(http://www.w3.org/Graphics/JPEG/itu-t81.pdf)

The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.
The modules described in this chapter provide interfaces to features that are unique to the SunOS operating system (versions 4 and 5; the latter is also known as Solaris version 2).

19.1 Sunaudiodev — Access to Sun audio hardware

This module allows you to access the Sun audio interface. The Sun audio hardware is capable of recording and playing back audio data in u-LAW format with a sample rate of 8K per second. A full description can be found in the audio(7I) manual page.

The module SUNAUDIODEV defines constants which may be used with this module.

This module defines the following variables and functions:

**exception error**
This exception is raised on all errors. The argument is a string describing what went wrong.

**open**(mode)
This function opens the audio device and returns a Sun audio device object. This object can then be used to do I/O on. The *mode* parameter is one of 'r' for record-only access, 'w' for play-only access, 'rw' for both and 'control' for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See audio(7I) for details.

As per the manpage, this module first looks in the environment variable AUDIODEV for the base audio device filename. If not found, it falls back to '/dev/audio'. The control device is calculated by appending "ctl" to the base audio device.

19.1.1 Audio Device Objects

The audio device objects are returned by open() define the following methods (except control objects which only provide getinfo(), setinfo(), fileno(), and drain()):

**close()**
This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

**fileno()**
Returns the file descriptor associated with the device. This can be used to set up SIGPOLL notification, as described below.

**drain()**
This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.
flush()  
This method discards all pending output. It can be used avoid the slow response to a user's stop request (due to buffering of up to one second of sound).

getinfo()  
This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a number of attributes describing the current device status. The names and meanings of the attributes are described in <sun/audioioio.h> and in the audio(7I) manual page. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the play substructure have 'o_' prepended to their name and members of the record structure have 'i_'. So, the C member play.sample_rate is accessed as o_sample_rate, record.gain as i_gain and monitor_gain plainly as monitor_gain.

ibufcount()  
This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a read() call of so many samples.

obufcount()  
This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

read(size)  
This method reads size samples from the audio input and returns them as a Python string. The function blocks until enough data is available.

setinfo(status)  
This method sets the audio device status parameters. The status parameter is an device status object as returned by getinfo() and possibly modified by the program.

write(samples)  
Write is passed a Python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

The audio device supports asynchronous notification of various events, through the SIGPOLL signal. Here's an example of how you might enable this in Python:

```python
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```

### 19.2 SUNAUDIODEV — Constants used with sunaudiodev

This is a companion module to sunaudiodev which defines useful symbolic constants like MIN_GAIN, MAX_GAIN, SPEAKER, etc. The names of the constants are the same names as used in the C include file <sun/audioioio.h>, with the leading string 'AUDIO_' stripped.
CHAPTER

TWENTY

MS Windows Specific Services

This chapter describes modules that are only available on MS Windows platforms.

- **msvcrt**: Miscellaneous useful routines from the MS VC++ runtime.
- **_winreg**: Routines and objects for manipulating the Windows registry.
- **winsound**: Access to the sound-playing machinery for Windows.

### 20.1 msvcrt – Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the `getpass` module uses this in the implementation of the `getpass()` function.

Further documentation on these functions can be found in the Platform API documentation.

#### 20.1.1 File Operations

- **locking** *(fd, mode, nbytes)*
  
  Lock part of a file based on file descriptor *fd* from the C runtime. Raises `IOError` on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

  - **LK_LOCK**
  - **LK_RLCK**
    
    Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, `IOError` is raised.

  - **LK_NBLCK**
  - **LK_NBRLCK**
    
    Locks the specified bytes. If the bytes cannot be locked, `IOError` is raised.

  - **LK_UNLCK**
    
    Unlocks the specified bytes, which must have been previously locked.

- **setmode** *(fd, flags)*
  
  Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be `os.O_TEXT`; for binary, it should be `os.O_BINARY`.

- **open_osfhandle** *(handle, flags)*
  
  Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bit-wise OR of `os.O_APPEND`, `os.O_RDONLY`, and `os.O_TEXT`. The returned file descriptor may be used as a parameter to `os.fdopen()` to create a file object.
get_osfhandle (fd)
Return the file handle for the file descriptor fd. Raises IOError if fd is not recognized.

20.1.2 Console I/O

kbhit()
Return true if a keypress is waiting to be read.

getch()
Read a keypress and return the resulting character. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for Enter to be pressed. If the pressed key was a special function key, this will return \000 or \xe0; the next call will return the keycode. The Control-C keypress cannot be read with this function.

getche()
Similar to getch(), but the keypress will be echoed if it represents a printable character.

putch (char)
Print the character char to the console without buffering.

ungetch (char)
Cause the character char to be “pushed back” into the console buffer; it will be the next character read by getch() or getche().

20.1.3 Other Functions

heapmin()
Force the malloc() heap to clean itself up and return unused blocks to the operating system. This only works on Windows NT. On failure, this raises IOError.

20.2 _winreg – Windows registry access

New in version 2.0.

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

This module exposes a very low-level interface to the Windows registry; it is expected that in the future a new winreg module will be created offering a higher-level interface to the registry API.

This module offers the following functions:

CloseKey (hkey)
Closes a previously opened registry key. The hkey argument specifies a previously opened key.

Note that if hkey is not closed using this method, (or the handle.Close() closed when the hkey object is destroyed by Python.

ConnectRegistry (computer_name, key)
Establishes a connection to a predefined registry handle on another computer, and returns a handle object
computer_name is the name of the remote computer, of the form ‘:\\computername’. If None, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an EnvironmentError exception is raised.
CreateKey (key, sub_key)
Creates or opens the specified key, returning a handle object

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

If key is one of the predefined keys, sub_key may be None. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key

The return value is the handle of the opened key. If the function fails, an EnvironmentError exception is raised.

DeleteKey (key, sub_key)
Deletes the specified key.

key is an already open key, or any one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the key parameter. This value must not be None, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an EnvironmentError exception is raised.

DeleteValue (key, value)
Removes a named value from a registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value is a string that identifies the value to remove.

EnumKey (key, index)
Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or any one of the predefined HKEY_* constants.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an EnvironmentError exception is raised, indicating, no more values are available.

EnumValue (key, index)
Enumerates values of an open registry key, returning a tuple.

key is an already open key, or any one of the predefined HKEY_* constants.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an EnvironmentError exception is raised, indicating no more values.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A string that identifies the value name</td>
</tr>
<tr>
<td>1</td>
<td>An object that holds the value data, and whose type depends on the underlying registry type</td>
</tr>
<tr>
<td>2</td>
<td>An integer that identifies the type of the value data</td>
</tr>
</tbody>
</table>

FlushKey (key)
Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined HKEY_* constants.

It is not necessary to call RegFlushKey to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike CloseKey(), the
FlushKey() method returns only when all the data has been written to the registry. An application should only call FlushKey() if it requires absolute certainty that registry changes are on disk.

*If you don’t know whether a FlushKey() call is required, it probably isn’t.*

**RegLoadKey** *(key, sub_key, file_name)*

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

- **key** is an already open key, or any of the predefined HKEY_ constants.
- **sub_key** is a string that identifies the sub_key to load
- **file_name** is the name of the file to load registry data from. This file must have been created with the SaveKey() function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to LoadKey() fails if the calling process does not have the SE_RESTORE_PRIVILEGE privilege. Note that privileges are different than permissions - see the Win32 documentation for more details.

If **key** is a handle returned by **ConnectRegistry()**, then the path specified in **fileName** is relative to the remote computer.

The Win32 documentation implies **key** must be in the HKEY_USER or HKEY_LOCAL_MACHINE tree. This may or may not be true.

**OpenKey** *(key, sub_key[, res = 0 ][, sam = KEY_READ ])*

Opens the specified key, returning a handle object

- **key** is an already open key, or any one of the predefined HKEY_ constants.
- **sub_key** is a string that identifies the sub_key to open
- **res** is a reserved integer, and must be zero. The default is zero.
- **sam** is an integer that specifies an access mask that describes the desired security access for the key. Default is KEY_READ

The result is a new handle to the specified key

If the function fails, **EnvironmentError** is raised.

**OpenKeyEx()**

The functionality of **OpenKeyEx()** is provided via **OpenKey()**, by the use of default arguments.

**QueryInfoKey** *(key)*

Returns information about a key, as a tuple.

- **key** is an already open key, or one of the predefined HKEY_ constants.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An integer giving the number of sub keys this key has.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the number of values this key has.</td>
</tr>
<tr>
<td>2</td>
<td>A long integer giving when the key was last modified (if available) as 100’s of nanoseconds since Jan 1, 1600.</td>
</tr>
</tbody>
</table>

**QueryValue** *(key, sub_key)*

Retrieves the unnamed value for a key, as a string

- **key** is an already open key, or one of the predefined HKEY_ constants.
- **sub_key** is a string that holds the name of the subkey with which the value is associated. If this parameter is None or empty, the function retrieves the value set by the **SetValue()** method for the key identified by **key**.

Values in the registry have name, type, and data components. This method retrieves the data for a key’s first value that has a NULL name. But the underlying API call doesn’t return the type, Lame Lame Lame, DO NOT USE THIS!!!
**QueryValueEx** *(key, value_name)*

Retrieves the type and data for a specified value name associated with an open registry key.

*key* is an already open key, or one of the predefined HKEY_* constants.  
*value_name* is a string indicating the value to query.

The result is a tuple of 2 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The value of the registry item.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the registry type for this value.</td>
</tr>
</tbody>
</table>

**SaveKey** *(key, file_name)*

Saves the specified key, and all its subkeys to the specified file.

*key* is an already open key, or one of the predefined HKEY_* constants.  
*file_name* is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the LoadKey(), ReplaceKey() or RestoreKey() methods.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the SeBackupPrivilege security privilege. Note that privileges are different than permissions - see the Win32 documentation for more details.

This function passes NULL for security_attributes to the API.

**SetValue** *(key, sub_key, type, value)*

Associates a value with a specified key.

*key* is an already open key, or one of the predefined HKEY_* constants.  
*sub_key* is a string that names the subkey with which the value is associated.  
*type* is an integer that specifies the type of the data. Currently this must be REG_SZ, meaning only strings are supported. Use the SetValueEx() function for support for other data types.  
*value* is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with KEY_SET_VALUE access.

**SetValueEx** *(key, value_name, reserved, type, value)*

Stores data in the value field of an open registry key.

*key* is an already open key, or one of the predefined HKEY_* constants.  
*sub_key* is a string that names the subkey with which the value is associated.  
*type* is an integer that specifies the type of the data. This should be one of the following constants defined in this module:
### Constant | Meaning
---|---
REG_BINARY | Binary data in any form.
REG_DWORD | A 32-bit number.
REG_DWORD_LITTLE_ENDIAN | A 32-bit number in little-endian format.
REG_DWORD_BIG_ENDIAN | A 32-bit number in big-endian format.
REG_EXPAND_SZ | Null-terminated string containing references to environment variables (\%PATH\%).
REG_LINK | A Unicode symbolic link.
REG_MULTI_SZ | A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)
REG_NONE | No defined value type.
REG_RESOURCE_LIST | A device-driver resource list.
REG_SZ | A null-terminated string.

*reserved* can be anything - zero is always passed to the API.

*value* is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with *KEY_SET_VALUE* access.

To open the key, use the `CreateKeyEx()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

#### 20.2.1 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the `Close()` method on the object, or the `CloseKey()` function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__nonzero__()` - thus

```python
    if handle:
        print "Yes"
```

will print *Yes* if the handle is currently valid (i.e., has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in `int()` function, in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

`Close()`

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

`Detach()`

Detaches the Windows handle from the handle object.

The result is an integer (or long on 64 bit Windows) that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.
After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

20.3  winsound — Sound-playing interface for Windows

New in version 1.5.2.

The winsound module provides access to the basic sound-playing machinery provided by Windows platforms. It includes two functions and several constants.

**Beep** *(frequency, duration)*

Beep the PC’s speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, **RuntimeError** is raised. **Note:** Under Windows 95 and 98, the Windows `Beep()` function exists but is useless (it ignores its arguments). In that case Python simulates it via direct port manipulation (added in version 2.1). It’s unknown whether that will work on all systems. New in version 1.6.

**PlaySound** *(sound, flags)*

Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, audio data as a string, or None. Its interpretation depends on the value of *flags*, which can be a bit-wise ORed combination of the constants described below. If the system indicates an error, **RuntimeError** is raised.

**SND_FILENAME**

The *sound* parameter is the name of a WAV file. Do not use with **SND_ALIAS**.

**SND_ALIAS**

The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless **SND_NODEFAULT** is also specified. If no default sound is registered, raise **RuntimeError**. Do not use with **SND_FILENAME**.

All Win32 systems support at least the following; most systems support many more:

<table>
<thead>
<tr>
<th>PlaySound() <em>name</em></th>
<th>Corresponding Control Panel Sound name</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SystemAsterisk'</td>
<td>Asterisk</td>
</tr>
<tr>
<td>'SystemExclamation'</td>
<td>Exclamation</td>
</tr>
<tr>
<td>'SystemExit'</td>
<td>Exit Windows</td>
</tr>
<tr>
<td>'SystemHand'</td>
<td>Critical Stop</td>
</tr>
<tr>
<td>'SystemQuestion'</td>
<td>Question</td>
</tr>
</tbody>
</table>

For example:

```python
import winsound

# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because "*") probably isn’t the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

**SND_LOOP**

Play the sound repeatedly. The **SND_ASYNC** flag must also be used to avoid blocking. Cannot be used with **SND_MEMORY**.

**SND_MEMORY**

The *sound* parameter to `PlaySound()` is a memory image of a WAV file, as a string.

**Note:** This module does not support playing from a memory image asynchronously, so a combination of this flag and **SND_ASYNC** will raise **RuntimeError**.
**SND_PURGE**

Stop playing all instances of the specified sound.

**SND_ASYNC**

Return immediately, allowing sounds to play asynchronously.

**SND_NODEFAULT**

If the specified sound cannot be found, do not play the system default sound.

**SND_NOSTOP**

Do not interrupt sounds currently playing.

**SND_NOWAIT**

Return immediately if the sound driver is busy.
Undocumented Modules

Here’s a quick listing of modules that are currently undocumented, but that should be documented. Feel free to contribute documentation for them! (Send via email to python-docs@python.org.)

The idea and original contents for this chapter were taken from a posting by Fredrik Lundh; the specific contents of this chapter have been substantially revised.

A.1 Frameworks

Frameworks tend to be harder to document, but are well worth the effort spent.

Tkinter — Interface to Tcl/Tk for graphical user interfaces; Fredrik Lundh is working on this one! See An Introduction to Tkinter at http://www.pythonware.com/library.htm for on-line reference material.

TkDnd — Drag-and-drop support for Tkinter.

turtle — Turtle graphics in a Tk window.

test — Regression testing framework. This is used for the Python regression test, but is useful for other Python libraries as well. This is a package rather than a single module.

A.2 Miscellaneous useful utilities

Some of these are very old and/or not very robust; marked with “hmm.”

bdb — A generic Python debugger base class (used by pdb).

ihooks — Import hook support (for rexec; may become obsolete).

A.3 Platform specific modules

These modules are used to implement the os.path module, and are not documented beyond this mention. There’s little need to document these.

dospath — Implementation of os.path on MS-DOS.

ntpath — Implementation on os.path on Win32, Win64, WinCE, and OS/2 platforms.

posixpath — Implementation on os.path on POSIX.
A.4 Multimedia

- **audiodev** — Platform-independent API for playing audio data.
- **sunaudio** — Interpret Sun audio headers (may become obsolete or a tool/demo).
- **toaiff** — Convert “arbitrary” sound files to AIFF files; should probably become a tool or demo. Requires the external program *sox*.

A.5 Obsolete

These modules are not normally available for import; additional work must be done to make them available.

Those which are written in Python will be installed into the directory ‘lib-old’ installed as part of the standard library. To use these, the directory must be added to *sys.path*, possibly using PYTHONPATH.

Obsolete extension modules written in C are not built by default. Under UNIX, these must be enabled by uncommenting the appropriate lines in ‘Modules/Setup’ in the build tree and either rebuilding Python if the modules are statically linked, or building and installing the shared object if using dynamically-loaded extensions.

- **addpack** — Alternate approach to packages. Use the built-in package support instead.
- **cmp** — File comparison function. Use the newer *filecmp* instead.
- **cmpcache** — Caching version of the obsolete cmp module. Use the newer *filecmp* instead.
- **codehack** — Extract function name or line number from a function code object (these are now accessible as attributes: *co.co_name*, *func.func_name*, *co.co_firstlineno*).
- **dircmp** — Class to build directory diff tools on (may become a demo or tool). **Deprecated since release 2.0.** The *filecmp* module replaces dircmp.
- **dump** — Print python code that reconstructs a variable.
- **fmt** — Text formatting abstractions (too slow).
- **lockfile** — Wrapper around FCNTL file locking (use *fcntl.lockf()*/flock() instead; see *fcntl*).
- **newdir** — New dir() function (the standard dir() is now just as good).
- **Pra** — Helper for fmt.
- **poly** — Polynomials.
- **regex** — Emacs-style regular expression support; may still be used in some old code (extension module). Refer to the Python 1.6 Documentation for documentation.
- **regsub** — Regular expression based string replacement utilities, for use with regex (extension module). Refer to the Python 1.6 Documentation for documentation.
- **tb** — Print tracebacks, with a dump of local variables (use pdb.pm() or *traceback* instead).
- **timing** — Measure time intervals to high resolution (use time.clock() instead). (This is an extension module.)
- **tzparse** — Parse a timezone specification (unfinished; may disappear in the future, and does not work when the TZ environment variable is not set).
- **util** — Useful functions that don’t fit elsewhere.
- **whatsound** — Recognize sound files; use *sndhdr* instead.
zmod — Compute properties of mathematical “fields.”

The following modules are obsolete, but are likely to re-surface as tools or scripts:

find — Find files matching pattern in directory tree.
grep — grep implementation in Python.
packmail — Create a self-unpacking UNIX shell archive.

The following modules were documented in previous versions of this manual, but are now considered obsolete. The source for the documentation is still available as part of the documentation source archive.

ni — Import modules in “packages.” Basic package support is now built in. The built-in support is very similar to what is provided in this module.
rand — Old interface to the random number generator.
soundex — Algorithm for collapsing names which sound similar to a shared key. The specific algorithm doesn’t seem to match any published algorithm. (This is an extension module.)

A.6 SGI-specific Extension modules

The following are SGI specific, and may be out of touch with the current version of reality.

cl — Interface to the SGI compression library.
sv — Interface to the “simple video” board on SGI Indigo (obsolete hardware).
Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

All bug reports should be submitted via the Python Bug Tracker on SourceForge (http://sourceforge.net/bugs/?group_id=5470). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

Before submitting a report, please log into SourceForge if you are a member; this will make it possible for the developers to contact you for additional information if needed. If you are not a SourceForge member but would not mind the developers contacting you, you may include your email address in your bug description. In this case, please realize that the information is publicly available and cannot be protected.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box near the bottom of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker (http://sourceforge.net/bugs/?group_id=5470). Select the “Submit a Bug” link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the “Summary” and “Details” fields. For the summary, enter a very short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the “Category” field, which allows you to place the bug report into a broad category (such as “Documentation” or “Library”).

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. If you have a SourceForge account and logged in to report the problem, you will receive an update each time action is taken on the bug.

See Also:

  
  Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

  
  Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
MODULE INDEX

Symbols
__builtins__, 75
__main__, 75
_winreg, 460

A
aifc, 395
AL, 441
al, 439
anydbm, 225
array, 126
asyncore, 323
atexit, 40
audioop, 391

B
base64, 345
BaseHTTPServer, 315
Bastion, 416
binascii, 339
binhex, 338
bisect, 125
bsddb, 228

calendar, 133
cd, 441
cgi, 277
CGIHTTPServer, 318
chunk, 401
cmath, 121
cmd, 134
code, 67
codecs, 96
codeop, 69
colorsys, 402
commands, 255
compileall, 429
ConfigParser, 129
Cookie, 319
copy, 60
copy_reg, 59
cPickle, 59
crypt, 242
cStringIO, 95
curses, 164
curses.ascii, 181
curses.panel, 183
curses.textpad, 179
curses.wrapper, 180

d
dbhash, 226
dbm, 243
DEVICE, 453
difflib, 90
dircache, 152
dis, 430
dl, 242
doctest, 103
doctest, 103
dumbdbm, 226

e
errno, 186
exceptions, 16

f
fcntl, 248
filecmp, 156
fileinput, 131
FL, 449
f1, 445
flp, 450
fm, 450
fnmatch, 193
formatter, 327
fpectl, 39
fpformat, 94
ftplib, 295

g
gc, 34
gdbm, 244
getopt, 184
getpass, 164
gettext, 199
GL, 453
glob, 192
gopherlib, 298
glp, 241
gzip, 231
htmlentitydefs, 357
httplib, 293
imageop, 394
imaplib, 300
imghdr, 404
imp, 64
inspect, 48
jpeg, 454
keyword, 427
linecache, 53
locale, 195
mailbox, 346
mailcap, 343
marshal, 61
math, 119
md5, 407
mhlib, 347
mimetypes, 344
MimeWriter, 335
mimify, 349
mmap, 224
mpz, 409
msvcr, 459
multifile, 336
mutex, 164
netrc, 350
new, 73
nis, 254
ntplib, 303
operator, 44
os, 139
os.path, 150
parser, 417
pdb, 257
pickle, 54
pipes, 249
popen2, 158
poplib, 298
posix, 239
posixfile, 250
pprint, 69
profile, 266
pstats, 267
pty, 247
pwd, 240
py_compile, 429
pyclbr, 428
Queue, 223
quopri, 346
random, 122
re, 80
readline, 235
repr, 71
resource, 252
reexec, 414
rfc822, 331
rgbimg, 403
rlcompleter, 236
robotparser, 351
roto, 410
sched, 162
select, 214
sgmlib, 353
sha, 408
shelve, 59
shlex, 136
shutil, 193
signal, 207
SimpleHTTPServer, 318
site, 73
smtplib, 306
sndhdr, 404
socket, 209
SocketServer, 313
stat, 153
statcache, 155
statvfs, 156
string, 77
StringIO, 95
struct, 88
sunau, 397
SUNAUDIODEV, 458
sunaudiodev, 457
symbol, 426
sys, 29
syslog, 255

T
tabnanny, 428
telnetlib, 309
tempfile, 186
TERMIOS, 247
termios, 245
thread, 215
threading, 217
time, 159
token, 427
tokenize, 427
traceback, 52
tty, 247
types, 41

U
unicodedata, 100
unittest, 110
urllib, 283
urllib2, 287
urlparse, 312
user, 74
UserDict, 43
UserList, 43
UserString, 44
uu, 339

W
warnings, 62
wave, 399
weakref, 35
webbrowser, 275
whichdb, 227
whrandom, 125
winsound, 465

X
xdrlib, 341

xml.dom, 363
xml.dom.minidom, 373
xml.dom.pulldom, 377
xml.parsers.expat, 357
xml.sax, 377
xml.sax.handler, 379
xml.sax.saxutils, 383
xml.sax.xmlreader, 383
xmllib, 387
xreadlines, 132

Z
zipfile, 232
zlib, 229
Symbols

.ini
  file, 129
 pdbrc
  file, 259
 .pythonrc.py
  file, 74
 ==
  operator, 4
 __abs__ () (in module operator), 45
 __add__ () (AddressList method), 333
 __add__ () (in module operator), 44
 __and__ () (in module operator), 45
 __bases__ (class attribute), 16
 __builtin__ (built-in module), 75
 __class__ (instance attribute), 16
 __cmp__ () (instance method), 4
 __concat__ () (in module operator), 45
 __contains__ () (in module operator), 46
 __copy__ () (copy protocol), 61
 __deepcopy__ () (copy protocol), 61
 __delitem__ () (in module operator), 46
 __delislice__ () (in module operator), 46
 __dict__ (instance attribute), 55
 __dict__ (object attribute), 16
 __displayhook__ (data in sys), 30
 __div__ () (in module operator), 45
 __excepthook__ (data in sys), 30
 __getinitargs__ () (copy protocol), 55, 61
 __getitem__ () (in module operator), 46
 __getslice__ () (in module operator), 46
 __getstate__ () (copy protocol), 55, 61
 __ladd__ () (AddressList method), 333
 __import__ () (in module), 20
 __init__ () (NullTranslations method), 200
 __init__ () (instance constructor), 55
 __invert__ () (in module operator), 45
 __invert__ () (in module operator), 45
 __isub__ () (AddressList method), 334
 __len__ () (AddressList method), 333
 __lshift__ () (in module operator), 45
 __main__ (built-in module), 75
 __members__ (object attribute), 16
 __methods__ (object attribute), 16
 __mod__ () (in module operator), 45
 __mul__ () (in module operator), 45
 __neg__ () (in module operator), 45
 __not__ () (in module operator), 45
 __or__ () (in module operator), 45
 __pos__ () (in module operator), 45
 __repeat__ () (in module operator), 46
 __repr__ () (ntrc method), 351
 __rshift__ () (in module operator), 45
 __setitem__ () (in module operator), 46
 __setslice__ () (in module operator), 46
 __setstate__ () (copy protocol), 55, 61
 __stderr__ (data in sys), 33
 __stdin__ (data in sys), 33
 __stdout__ (data in sys), 33
 __str__ () (AddressList method), 333
 __sub__ () (AddressList method), 334
 __sub__ () (in module operator), 44
 __xor__ () (in module operator), 45
 __exit__ () (in module os), 147
 __getframe__ () (module sys), 31
 __locale__ (built-in module), 195
 __parse__ () (NullTranslations method), 200
 __winreg__ (extension module), 460

A
A-LAW, 396, 405
a2b_base64 () (module binascii), 339
a2b_hex () (module binascii), 340
a2b_hqx () (module binascii), 340
a2b_uu () (module binascii), 339
ABC language, 4
abort ()
  FTP method, 296
  in module os, 147
above () (method), 183
abs ()
  in module, 21
  in module operator, 45

477
absopath (in module os.path), 150
AbstractBasicAuthHandler (class in urllib2), 288
AbstractDigestAuthHandler (class in urllib2), 288
AbstractFormatter (class in formatter), 329
AbstractWriter (class in formatter), 331
accept()
    dispatcher method, 324
    socket method, 212
accept2dyear (data in time), 160
access (in module os), 144
acos (in module cmath), 121
acos (in module math), 119
acosh (in module cmath), 121
acquire()
    Condition method, 220
    lock method, 216
    Semaphore method, 220
    Thread method, 218
activate_form () (form method), 447
activateCount () (in module threading), 217
add()
    in module audioop, 391
    in module operator, 44
    Stats method, 267
add_box () (form method), 447
add_browser () (form method), 448
add_button () (form method), 447
add_choice () (form method), 448
add_clock () (form method), 447
add_counter () (form method), 448
add_data () (Request method), 289
add_dial () (form method), 447
add_flowing_data () (formatter method), 328
add_handler () (OpenerDirector method), 290
add_header () (Request method), 289
add_hor_rule () (formatter method), 328
add_input () (form method), 448
add_label_data () (formatter method), 328
add_lightbutton () (form method), 447
add_line_break () (formatter method), 328
add_literal_data () (formatter method), 328
add_menu () (form method), 448
add_parent () (BaseHandler method), 290
add_password () (HTTPPasswordMgr method), 291
add_positioner () (form method), 447
add_roundbutton () (form method), 447
add_section () (ConfigParser method), 130
add_slider () (form method), 447
add_text () (form method), 447
add_timer () (form method), 448
add_valslider () (form method), 447
addcallback () (CD parser method), 444
addch () (method), 170
addError () (TestResult method), 118
addFailure () (TestResult method), 118
addheader () (MimeWriter method), 335
addnstr () (method), 170
address_family (data in SocketServer), 314
address_string () (BaseHTTPRequestHandler method), 317
AddressList (class in rfc822), 331
addresslist (AddressList attribute), 334
addstr () (method), 170
addSuccess () (TestResult method), 118
addTest () (TestSuite method), 117
addTests () (TestSuite method), 117
adler32 () (in module zlib), 230
ADPCM, Intel/DVI, 391
adpcm2lin () (in module audioop), 391
adpcm32lin () (in module audioop), 391
AF_INET (data in socket), 210
AF_UNIX (data in socket), 210
aifc () (aifc method), 396
aifc (standard module), 395
AIFF, 395, 401
aiff () (aifc method), 396
AIFF-C, 395, 401
AL (standard module), 439, 441
al (built-in module), 439
alarm () (in module signal), 208
all_errors (data in ftplib), 295
all_features (data in xml.sax.handler), 380
all_properties (data in xml.sax.handler), 380
allocate_lock () (in module thread), 216
allow_reuse_address (data in SocketServer), 314
allowremoval () (CD player method), 443
alt () (in module curses.ascii), 182
altsep (data in os), 150
altzone (data in time), 160
anchor_bgn () (HTMLParser method), 356
anchor_end () (HTMLParser method), 356
and
    operator, 3, 4
and () (in module operator), 45
annotate () (in module dircache), 152
anydbm (standard module), 225
apop () (POP3 method), 299
append ()
    array method, 127
    IMAP4 method, 301
    list method, 11
    Template method, 249
appendChild () (Node method), 366
basename() (in module os.path), 150
 Bastion() (in module Bastion), 416
 Bastion (standard module), 416
 BastionClass (class in Bastion), 416
 baudrate() (in module curses), 165
 bdb (standard module), 257
 Beep() (in module winsound), 465
 beep() (in module curses), 165
 below() (method), 183
 benchmarking, 160
 bestreadsize() (CD player method), 443
 betavariate() (in module random), 124
 bgn_group() (form method), 447
 bias() (in module audioop), 392
 bidirectional() (in module unicodedata), 100
 binary
 data, packing, 88
 binary() (mpz method), 410
 binary semaphores, 216
 binascii (built-in module), 339
 bind()
 dispatcher method, 324
 socket method, 212
 bindtextdomain() (in module gettext), 199
 binhex() (in module binhex), 338
 binhex (standard module), 338, 339
 bisect() (in module bisect), 126
 bisect (standard module), 125
 bisect_left() (in module bisect), 126
 bisect_right() (in module bisect), 126
 bit-string
 operations, 6
 bkgd() (method), 171
 bkgdset() (method), 171
 BLOCKSIZE (data in cd), 442
 blocksize (data in sha), 408
 body() (NNTPDataError method), 306
 BOM (data in codecs), 96
 BOM32_BE (data in codecs), 97
 BOM32_LE (data in codecs), 97
 BOM64_BE (data in codecs), 97
 BOM64_LE (data in codecs), 97
 BOM_BE (data in codecs), 97
 BOM_LE (data in codecs), 97
 Boolean
 operations, 3, 4
 type, 3
 border() (method), 171
 bottom() (method), 183
 bottom_panel() (in module curses.panel), 183
 box() (method), 171
 BROWSER, 275, 276
 bsddb
 built-in module, 225, 226
 extension module, 228
 btopen() (in module bsddb), 228
 buffer
 object, 6
 buffer()
 built-in function, 6, 43
 in module, 21
 buffer size, I/O, 25
 buffer_info() (array method), 127
 BufferType (data in types), 43
 build_opener() (in module urllib2), 287
 built-in
 exceptions, 3
 functions, 3
 types, 3
 builtin_module_names (data in sys), 30
 BuiltinFunctionType (data in types), 42
 BuiltinMethodType (data in types), 42
 byte-code
 file, 64, 66, 429
 byteorder (data in sys), 30
 byteswap() (array method), 127
 C
 C language, 4, 5
 structures, 88
 C_BUILTIN (data in imp), 65
 C_EXTENSION (data in imp), 65
 CacheFTPHandler (class in urllib2), 289
 calcsize() (in module struct), 88
 calendar() (in module calendar), 134
 calendar (standard module), 133
 call() (method), 243
 callable() (in module), 21
 CallableProxyType (data in weakref), 36
 can_change_color() (in module curses), 165
 can_fetch() (RobotFileParser method), 351
 cancel() (scheduler method), 163
 capitalize()
 in module string, 78
 string method, 7
 capwords() (in module string), 78
 cat() (in module nis), 254
 catalog (data in cd), 442
 category() (in module unicodedata), 100
 cbreak() (in module unicodedata), 100
 cd (built-in module), 441
 CDROM (data in cd), 442
 ceil()
 in module math, 119
 in module math, 5
 center()
 in module string, 79
string method, 7

CGI
protocol, 277
cgi (standard module), 277
cgi_directories (CGIHTTPRequestHandler attribute), 319
CGIHTTPRequestHandler (class in CGIHTTPServer), 318
CGIHTTPServer (standard module), 315, 318
chaining
comparisons, 4
CHAR_MAX (data in locale), 197
character, 100
CharacterDataHandler() (xmlparser method), 360
characters() (ContentHandler method), 381
 CHARSET (data in mimify), 350
charset() (NullTranslations method), 201
chdir() (in module os), 144
check()
IMAP4 method, 301
in module tabnanny, 428
check_forms() (in module fl), 445
checkcache() (in module linecache), 54
checksum
Cyclic Redundancy Check, 230
MD5, 407
SHA, 408
childerr (Popen4 attribute), 159
childNodes (Node attribute), 366
chmod() (in module os), 144
choice()
in module random, 123
in module wrandom, 125
choose_boundary() (in module mimetools), 334
chown() (in module os), 145
chr() (in module ), 21
Chunk (class in chunk), 402
chunk (standard module), 401
cipher
DES, 242, 407
Enigma, 410
IDEA, 407
classobj() (in module new), 73
ClassType (data in types), 42
clear()
method, 171
dictionary method, 11
Event method, 221
clearcache() (in module linecache), 54
clearok() (method), 171
client_address (BaseHTTPRequestHandler attribute), 316
clock() (in module time), 160
close() (Template method), 249
clonenode() (method), 374
Node method, 367
Close() (method), 464
close()
method, 224, 228, 243
aifc method, 396, 397
AU_read method, 398
AU_write method, 399
audio device method, 457
BaseHandler method, 290
CD player method, 443
Chunk method, 402
dispatcher method, 324
file method, 14
FTP method, 298
IMAP4 method, 301
in module fileinput, 132
in module os, 142
IncrementalParser method, 385
OpenerDirector method, 290
SGMLParser method, 354
socket method, 212
StringIO method, 95
Telnet method, 311
Wave_read method, 400
Wave_write method, 401
XMLParser method, 388
ZipFile method, 233
closed (file attribute), 16
CloseKey() (in module _winreg), 460
closelog() (in module syslog), 255
closeport() (audio port method), 440
crltobot() (method), 171
crltoeol() (method), 171
cmath (built-in module), 121
Cmd (class in cmd), 134
cmd (standard module), 134, 257
cmdloop() (Cmd method), 134
cmp()
built-in function, 197
in module , 21
in module filecmp, 156
cmp_op (data in dis), 431
cmpfiles() (in module filecmp), 157
code
object, 13, 14, 61
code() (in module new), 73
code
ExpatError attribute, 361
standard module, 67
Codecs, 96
decode, 96
decode
  Codecs, 96
decode()
  method, 97
  in module base64, 345
  in module mimetools, 334
  in module quopri, 346
  in module uu, 339
decodestring() (in module base64), 345
decomposition() (in module unicodedata), 101
decompress()
  Decompress method, 231
  in module jpeg, 454
  in module zlib, 230
decompressobj() (in module zlib), 230
decrypt() (rotor method), 410
decryptmore() (rotor method), 410
deepcopy() (in copy), 60
defProgMode() (in module curses), 166
defShellMode() (in module curses), 166
default() (Cmd method), 135
default_bufsize (data in xml.dom.pulldom), 377
default_open() (BaseHandler method), 290
DefaultHandler() (xmlparser method), 360
DefaultHandlerExpand() (xmlparser method), 360
defaults() (ConfigParser method), 130
defaultTestLoader (data in unittest), 115
defaultTestResult() (TestCase method), 117
defpath (data in os), 150
de
statement, 11
delattr() (in module ), 22
delay_output () (in module curses), 166
delch() (method), 171
dele() (POP3 method), 299
delete()
  FTP method, 297
  IMAP4 method, 302
delete_object () (FORMS object method), 449
deletefolder() (MH method), 348
DeleteKey() (in module _winreg), 461
deletein() (method), 171
deleteparser() (CD parser method), 444
DeleteValue() (in module _winreg), 461
delitem() (in module operator), 46
delslice() (in module operator), 46
DeprecationWarning (exception in exceptions), 20
derwin() (method), 171
DES
  cipher, 242, 407
descriptor, file, 14

Detach() (method), 464
deterministic profiling, 263
DEVICE (standard module), 453
device
  Enigma, 410
dgettext () (in module gettext), 199
dictionary
  object, 11
type, operations on, 11
DictionaryType (data in types), 42
DictType (data in types), 42
diff_files (dircmp attribute), 158
difflib (standard module), 90
digest()
  md5 method, 408
  sha method, 408
digestsize (data in sha), 408
digit() (in module unicodedata), 100
digits (data in string), 77
dir()
  FTP method, 297
  in module , 22
dircache (standard module), 152
dircmp (class in filecmp), 157
directory
  changing, 144
  creating, 145
  deleting, 145, 194
  site-packages, 74
  site-python, 74
dirname() (in module os.path), 150
dis() (in module dis), 430
dis (standard module), 430
disable() (in module gc), 34
disassemble() (in module dis), 430
disco() (in module dir), 431
dispatcher (class in asyncore), 323
displayhook () (in module sys), 30
distb() (in module dis), 430
dither2grey2() (in module imageop), 395
dither2mono() (in module imageop), 394
div() (in module operator), 45
division
  integer, 5
  long integer, 5
divm() (in module mpz), 409
divmod() (in module ), 22
dl (extension module), 242
dlhandle (data in sys), 30
do_command() (Textbox method), 180
do_forms() (in module fl), 445
do_GET() (SimpleHTTPRequestHandler method), 318
Index 485

do_HEAD() (SimpleHTTPRequestHandler method), 318

do_POST() (CGIHTTPRequestHandler method), 319
doc_header (Cmd attribute), 135
docmd() (SMTP method), 308
docstrings, 421
doctest (standard module), 103

DOCTYPE declaration, 388
documentElement (Document attribute), 368
DOMEventStream (class in xml.dom.pulldom), 377

DOMException (exception in xml.dom), 371

DomstringSizeErr (exception in xml.dom), 371
done() (Unpacker method), 342

dotall (data in re), 84
doupdate() (in module curses), 166
drain() (audio device method), 457

DTDHandler (class in xml.sax.handler), 379
dumbdbm (standard module), 225, 226

DumbWriter (class in formatter), 331
dump() in module marshal, 62
dumps() in module marshal, 62
dup() in module os, 142
dup2() in module os, 142

DuplicateSectionError (exception in ConfigParser), 129

e

e data in cmath, 121
data in math, 120

e2BIG (data in errno), 187
eACCES (data in errno), 187
eADDRINUSE (data in errno), 191
eADDRNOTAVAL (data in errno), 191
eADV (data in errno), 187

eAFNOSUPPORT (data in errno), 191
eAGAIN (data in errno), 187
eALREADY (data in errno), 192

EBADE (data in errno), 189

EBADF (data in errno), 187

EBADFD (data in errno), 190

EBADMSG (data in errno), 190

EBADR (data in errno), 189

EBADRQC (data in errno), 189

EBADSLT (data in errno), 189

EBFONT (data in errno), 189

EBUSY (data in errno), 187

ECHILD (data in errno), 187
echo() (in module curses), 166

echochar() (method), 172

ECHRNG (data in errno), 188

ECOMM (data in errno), 190

ECREAD (data in errno), 191

ECRREJ (data in errno), 192

ECRSET (data in errno), 191

EDeadLk (data in errno), 188

EDeadlock (data in errno), 189

EDestAddrReq (data in errno), 191

edit() (Textbox method), 179

EDOM (data in errno), 188

EDOTDOT (data in errno), 190

EDQUOT (data in errno), 192

EEXIST (data in errno), 187

EFileUse (data in errno), 187

EFBIG (data in errno), 188

ehlo() (SMTP method), 308

EHOSTDOWN (data in errno), 192

EHOSTUNREACH (data in errno), 192

EIDRM (data in errno), 188

EILSEQ (data in errno), 190

EINPROGRESS (data in errno), 192

EINTR (data in errno), 187

EINVAL (data in errno), 187

EIO (data in errno), 187

EISCONN (data in errno), 191

EISDIR (data in errno), 187

EISNAM (data in errno), 192
eject() (CD player method), 443

EL2HLT (data in errno), 189

EL2NSYNC (data in errno), 189

EL3HLT (data in errno), 189

EL3RST (data in errno), 189

ElementDeclHandler() (xmlparser method), 359

elements (XMLParser attribute), 388

ELIBACC (data in errno), 190

ELIBBAD (data in errno), 190

ELIBEXEC (data in errno), 190

ELIBMAX (data in errno), 190

ELIBSCN (data in errno), 190

Ellinghouse, Lance, 339, 410

EllipsisType (data in types), 43

ELNRNG (data in errno), 189

ELOOP (data in errno), 188

EMFILE (data in errno), 188

ENOMEM (data in errno), 188

Empty (exception in Queue), 223

empty() Queue method, 223
scheduler method, 163
codecs, 96
encode() (method), 97
in module base64, 346
in module mimetools, 334
in module quopri, 346
in module uu, 339
string method, 7
EncodedFile() (in module codecs), 96
codecstr() (module base64), 346
coding
base64, 345
quoted-printable, 346
encodings_map (in module mimetypes), 345
cipher, 410
device, 410
ENOANO (data in errno), 189
ENOBUFFS (data in errno), 191
ENOCRED (data in errno), 189
ENODATA (data in errno), 189
ENODEV (data in errno), 187
ENOENT (data in errno), 187
ENOEXEC (data in errno), 187
ENOLCK (data in errno), 188
ENOLINK (data in errno), 190
ENOMEM (data in errno), 187
ENOMSG (data in errno), 188
ENONET (data in errno), 189
ENOPKG (data in errno), 189
ENOPROTOOPT (data in errno), 191
ENOSPC (data in errno), 188
ENOSR (data in errno), 189
ENOSTR (data in errno), 189
ENOSYS (data in errno), 188
ENOERR (data in errno), 187
ENOTCONN (data in errno), 191
ENOTDIR (data in errno), 187
ENOTEMPTY (data in errno), 188
ENOTFOUND (data in errno), 192
ENOSTATE (data in errno), 190
ENOTTY (data in errno), 188
ENOTUNIQ (data in errno), 190
enter() (scheduler method), 163
enterabs() (scheduler method), 163
entities (DocumentType attribute), 367
ENTITY declaration, 389
EntityDeclHandler () (xmlparser method), 360
entitydefs
data in htmlentitydefs, 357
XMLParser attribute, 388
EntityResolver (class in xml.sax.handler), 379
enumerate()
in module fm, 450
in module threading, 217
EnumKey() (module _winreg), 461
EnumValue() (in module _winreg), 461
environ
data in os, 140
data in posix, 240
environment variables
BROWSER, 275, 276
COLUMNS, 170
HOME, 75, 151
KDEDIR, 276
LANGUAGE, 199, 200
LANG, 195, 196, 199, 200
LC_ALL, 199, 200
LC_MESSAGES, 199, 200
Index 487
ESTALE (data in errno), 192
ESTRPIPE (data in errno), 190
ETIME (data in errno), 189
ETIMEDOUT (data in errno), 191
ETOOMANYREFS (data in errno), 191
ETXTBSY (data in errno), 188
EUCEDE (data in errno), 192
EUNATCH (data in errno), 189
EUSERS (data in errno), 190
eval() in module string, 78
Event() (in module threading), 217
Event (class in threading), 221
event scheduling, 162
EWOULDBLOCK (data in errno), 188
exc_info() (in module sys), 31
exc_traceback (data in sys), 31
exec_prefix (in sys), 31
except statement, 17
except () (in module sys), 30
except (exception in exceptions), 17
except built-in, 3
except exceptions (standard module), 16
EXDEV (data in errno), 187
f() in module math, 119
F_BAVAIL (data in statvfs), 156
F_BFREE (data in statvfs), 156
F_BLOCKS (data in statvfs), 156
F_BSIZE (data in statvfs), 156
F_FAVAIL (data in statvfs), 156
F_FFREE (data in statvfs), 156
F_FILES (data in statvfs), 156
F_FLAGS (data in statvfs), 156
F_FNAME (data in statvfs), 156
F_FNAME (data in os), 144
f() (in module math), 120
f() (TestCase method), 117
false, 3
FancyURLopener (class in urllib), 285
FCNTL (standard module), 248
fcntl() (in module fcntl), 248
fcntl() (built-in module), 15, 248
fdopen() (in module fcntl), 250
fdopen() (in module os), 142
feature_external_ges (data in xml.sax.handler), 379
feature_external_pes (data in xml.sax.handler), 380
feature_namespace_prefixes (data in xml.sax.handler), 379
feature_namespaces (data in xml.sax.handler), 379
feature_string_interning (data in xml.sax.handler), 379
feature_validation (data in xml.sax.handler), 379
feed()
  IncrementalParser method, 385
  SGMLParser method, 354
  XMLParser method, 388
fetch() (IMAP4 method), 302
file
  .ini, 129
  .pdbrc, 259
  .pythonrc.py, 74
  byte-code, 64, 66, 429
  configuration, 129
  copying, 193
  debugger configuration, 259
  large files, 240
  mime.types, 345
  object, 14
  path configuration, 74
  temporary, 186
  user configuration, 74
file() (in module posixfile), 251
file (class descriptor attribute), 429
file control
  UNIX, 248
file descriptor, 14
file name
  temporary, 186
file object
  POSIX, 250
file_offset (ZipInfo attribute), 235
file_open() (FileHandler method), 292
file_size (ZipInfo attribute), 235
filecmp (standard module), 156
FileHandler (class in urllib2), 289
FileInput (class in fileinput), 132
fileinput (standard module), 131
filename() (in module fileinput), 131
filename (ZipInfo attribute), 234
filename_only (data in tabnanny), 428
filenames
  pathname expansion, 192
  wildcard expansion, 193
fileno()
  audio device method, 457
  file method, 14
  in module SocketServer, 314
  socket method, 212
  Telnet method, 311
fileopen() (in module posixfile), 250
FileType (data in types), 42
filter()
  in module, 22
  in module curses, 166
filterwarnings() (in module warnings), 64
find()
  method, 224
  in module gettext, 200
  in module string, 78
  string method, 7
find_first() (form method), 447
find_last() (form method), 447
find_longest_match() (SequenceMatcher method), 92
find_module() (in module imp), 64
find_user_password() (HTTPPasswordMgr method), 291
findall()
  in module re, 85
  RegexObject method, 86
findfactor() (in module audioop), 392
findfit() (in module audioop), 392
findfont() (in module fm), 450
findmatch() (in module mailcap), 344
findmax() (in module audioop), 392
finish() (in module SocketServer), 315
finish_request() (in module SocketServer), 314
first()
  method, 229
  dbhash method, 227
firstChild (Node attribute), 366
firstkey() (in module gdbm), 245
firstweekday() (in module calendar), 133
fix() (in module fpformat), 94
FL (standard module), 449
fl (built-in module), 445
flag_bits (ZipInfo attribute), 234
flags() (in module posixfile), 250
flags (RegExObject attribute), 86
flash() (in module curses), 166
flattening
objects, 54
float()
    built-in function, 5, 78
    in module, 23
floating point
    literals, 5
    object, 5
FloatingPointError
    exception in exceptions, 18
    exception in fpectl, 39
FloatType (data in types), 42
flock()
    (in module fcntl), 248
floor()
    in module math, 120
    in module math, 5
flp (standard module), 450
flush()
    method, 224
    audio device method, 458
    Compress method, 230
    Decompress method, 231
    file method, 14
    writer method, 329
flush_space() (formatter method), 328
flushheaders() (MimeWriter method), 335
flushinp() (in module curses), 166
FlushKey() (in module _winreg), 461
fm (built-in module), 450
fmod() (in module math), 120
fnmatch() (in module fnmatch), 193
fnmatch (standard module), 193
fnmatchcase() (in module fnmatch), 193
Folder (class in mhlib), 347
Font Manager, IRIS, 450
fontpath() (in module fm), 450
forget() (in module statcache), 155
forget_dir() (in module statcache), 155
forget_except_prefix() (in module statcache), 155
forget_prefix() (in module statcache), 155
fork()
    in module os, 147
    in module pty, 247
forkpty() (in module os), 147
Formal Public Identifier, 388
format() (in module locale), 197
format_exception() (in module traceback), 53
format_exception_only() (in module traceback), 52
format_list() (in module traceback), 52
format_stack() (in module traceback), 53
format_tb() (in module traceback), 53
formatargspec() (in module inspect), 51
formatargvalues() (in module inspect), 51
formatter
    HTMLParser attribute, 356
    standard module, 327, 355
formatting, string, 9
formatwarning() (in module warnings), 64
FORMS Library, 445
fp (AddressList attribute), 333
fpathconf() (in module os), 142
fpectl (extension module), 39
fpformat (standard module), 94
frame
    object, 209
FrameType (data in types), 43
freeze_form() (form method), 447
freeze_object() (FORMS object method), 449
fexp() (in module math), 120
fromchild (Popen4 attribute), 159
fromfd() (in module socket), 211
fromfile() (array method), 127
fromlist() (array method), 128
fromstring() (array method), 128
fstat() (in module os), 143
fstatvfs() (in module os), 143
FTP
    protocol, 285, 295
    FTP (class in ftplib), 295
ftp_open() (FTPHandler method), 293
ftp_proxy, 284
FTPHandler (class in urllib2), 289
ftplib (standard module), 295
ftpmirror.py, 296
ftruncate() (in module os), 143
Full (exception in Queue), 223
full() (Queue method), 223
func_code (function object attribute), 14
function() (in module new), 14
functions
    built-in, 3
FunctionTestCase (class in unittest), 115
FunctionType (data in types), 42
funny_files (dircmp attribute), 158
G
G.722, 396
gamma() (in module random), 124
garbage (data in gc), 35
gather() (Textbox method), 180
gauss() (in module random), 124
gc (extension module), 34
gcd() (in module mpz), 409
gcdext() (in module mpz), 409
gdbm (built-in module), 60, 225, 244
get()
    AddressList method, 332
getfile() HTTP method, 294
in module inspect, 50
getfillable() (audio port method), 440
getfilled() (audio port method), 440
getfillpoint() (audio port method), 441
getchartingheader() (AddressList method), 332
getfloat() (ConfigParser method), 130
getfloatmax() (audio configuration method), 440
getfontinfo() (in module fm), 451
getfontname() (in module fm), 450
getfqdn() (in module socket), 210
getframe() (audio port method), 440
getframerate() (au method), 395
gethostbyaddr() (socket method), 211
gethostbyname() (in module socket), 141
gethostbyname_ex() (in module socket), 210
gethostname() (in module socket), 141
getid() (module os), 140
getgrall() (in module grp), 242
getgrgid() (in module grp), 241
getgrnam() (in module grp), 241
getgroups() (in module os.path), 141
getheader() (AddressList method), 332
gethostbyaddr() in module socket, 141
in module socket, 211
gethostbyname() (in module socket), 210
gethostbyname_ex() (in module socket), 210
gethostname() in module socket, 141
in module socket, 211
getinfo() audio device method, 458
ZipFile method, 233
getinnerframes() (in module inspect), 51
GetInputContext() (xmlparser method), 358
getint() (ConfigParser method), 130
getitem() (in module operator), 46
getkey() (method), 172
getlast() (Folder method), 349
getLength() (AttributesImpl method), 387
getline() (in module linecache), 53
getLineNumber() (Locator method), 386
getLocale() (in module locale), 196
getlogin() (in module os), 140
getmaintype() (Message method), 335
getmark() aifc method, 396
AU_read method, 398
Wave_read method, 400
getmaxyx() (method), 172
getmcolor() (in module fl), 446
getmembers() (in module inspect), 49
getMessage() (SAXException method), 378
getmessagefilename() (Folder method), 348
getmodule() (in module inspect), 50
getmoduleinfo() (in module inspect), 49
getmodulename() (in module inspect), 50
getmouse() (in module curses), 166
gmtime() (in module os.path), 151
getName() (Thread method), 222
getName() (Chunk method), 402
getNameByQName() (AttributesNSImpl method), 387
getNames() (AttributesImpl method), 387
getnamespace() (XMLParser method), 388
getnchannels() aifc method, 395
AU_read method, 398
Wave_read method, 400
getnframes() aifc method, 395
AU_read method, 398
Wave_read method, 400
g getopt() (in module getopt), 184
getopt (standard module), 184
GetoptError (exception in getopt), 184
getouterframes() (in module inspect), 51
getoutput() (in module commands), 256
getpagesize() (in module resource), 254
getparam() (Message method), 335
getparams() aifc method, 396
AU_read method, 398
in module al, 440
Wave_read method, 400
getparyx() (method), 172
getpass() (in module getpass), 164
getpass (standard module), 164
getpath() (MH method), 348
getpeername() (socket method), 212
getpgroup() (in module os), 141
getpid() (in module os), 141
getplist() (Message method), 335
getppid() (in module os), 141
getprofile() (MH method), 348
getproperty() (XMLReader method), 385
getprotobyname() (in module socket), 211
getPublicId() InputSource method, 386
Locator method, 386
getpwall() (in module pwd), 241
getpwnam() (in module pwd), 241
getpwuid() (in module pwd), 241
getQNameByN() (AttributesNSImpl method), 387
getQNames() (AttributesNSImpl method), 387
get_queuesize() (audio configuration method), 440
getrawheader() (AddressList method), 332
get_reursionlimit() (in module sys), 31
get_reply() (HTTP method), 294
getrlimit() (in module resource), 252
getsample() (in module audioop), 392
getsampwidth() (aifc method), 395
AU_read method, 398
Wave_read method, 400
getsequences() (Folder method), 349
getsequencesfilename() (Folder method), 348
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
getservbyname() (in module socket), 211
getsignal() (in module signal), 208
getsize() (Chunk method), 402
in module os.path, 151
getsizes() (in module imgfile), 453
getslice() (in module operator), 46
getsockname() (socket method), 212
getsockopt() (socket method), 212
getsource() (in module inspect), 51
getsourcefile() (in module inspect), 50
getsourcelines() (in module inspect), 51
getstate() (in module random), 123
handle_authentication_request() (AbstractBasicAuthHandler method), 292
handle_authentication_request() (AbstractDigestAuthHandler method), 292
handle_cdata() (XMLParser method), 389
handle_charref() SGMLParser method, 354
XMLParser method, 389
handle_close() (dispatcher method), 323
handle_comment() SGMLParser method, 354
XMLParser method, 389
handle_connect() (dispatcher method), 323
handle_data() SGMLParser method, 354
XMLParser method, 389
handle_decl() (SGMLParser method), 354
handle_doctype() (XMLParser method), 388
handle_endtag() SGMLParser method, 354
XMLParser method, 388
handle_entityref() (SGMLParser method), 354
handle_error() (in module SocketServer), 314
handle_except() (dispatcher method), 323
handle_image() (HTMLParser method), 356
handle.proc() (XMLParser method), 389
handle_read() (dispatcher method), 323
handle_request() (in module SocketServer), 314
handle_special() (XMLParser method), 389
handle_starttag() SGMLParser method, 354
XMLParser method, 388
handle_write() (dispatcher method), 323
handle_xml() (XMLParser method), 388
has_colors() (in module curses), 167
has.data() (Request method), 289
has_extn() (SMTP method), 308
has.ic() (in module curses), 167
has.il() (in module curses), 167
has.key() method, 228
dictionary method, 11
in module curses, 167
has_option() (ConfigParser method), 130
has_section() (ConfigParser method), 130
hasattr() (in module), 23
hasAttributes() (Node method), 366
hasChildNodes() (Node method), 366
hascompare (data in dis), 431
hasconst (data in dis), 431
hasFeature() (DOMImplementation method), 365
hash() (in module), 23
hashopen() (in module bsddb), 228
hasjabs (data in dis), 431
hasjrel (data in dis), 431
haslocal (data in dis), 431
hasname (data in dis), 431
head() (NNTPDataError method), 306
header_offset (ZipInfo attribute), 235
headers
MIME, 277, 345
headers
AddressList attribute, 333
BaseHTTPRequestHandler attribute, 316
heapmin() (in module msvcr32), 460
helo() (SMTP method), 308
help() (NNTPDataError method), 305
hex() (in module), 23
hexadecimal
literals, 5
hexbin() (in module binhex), 338
hexdigest() md5 method, 408
sha method, 408
hexdigits (data in string), 77
hexlify() (in module binascii), 340
hexversion (data in sys), 31
hidden() (method), 183
hide() (method), 183
hide_form() (form method), 446
hide_object() (FORMS object method), 449
HierarchyRequestErr (exception in xml.dom), 371
hline() (method), 172
hls_to_rgb() (in module colorsys), 403
HOME, 75, 151
hosts (netrc attribute), 351
hsv_to_rgb() (in module colorsys), 403
HTML, 285, 355
htmlentitydefs (standard module), 357
htmlib (standard module), 286, 353, 355
HTMLParser (class in htmlib), 327, 356
htonl() (in module socket), 211
htons() (in module socket), 211
HTTP
protocol, 277, 285, 293, 315
HTTP (class in httpib), 293
http_error.301() (HTTPRedirectHandler method), 291
http_error.302() (HTTPRedirectHandler method), 291
http_error.401() (HTTPBasicAuthHandler method), 292
http_error.401() (HTTPDigestAuthHandler method), 292

494 Index
http_error_407() (ProxyBasicAuthHandler method), 292
http_error_407() (ProxyDigestAuthHandler method), 292
http_error_default() (BaseHandler method), 291
http_open() (HTTPHandler method), 292
http_proxy, 284
HTTPBasicAuthHandler (class in urllib2), 288
httpd, 315
HTTPDefaultErrorHandler (class in urllib2), 288
HTTPDigestAuthHandler (class in urllib2), 288
HTTPHandler (class in urllib2), 289
httplib (standard module), 293
HTTPPasswordMgr (class in urllib2), 288
HTTPPasswordMgrWithDefaultRealm (class in urllib2), 288
HTTPRedirectHandler (class in urllib2), 288
https_open() (HTTPSHandler method), 292
HTTPServer (class in BaseHTTPServer), 315
HTTPHandler (class in urllib2), 289
hypertext, 355
hypot() (in module math), 120
I (data in re), 84
I/O control
  buffering, 25, 142, 213
  POSIX, 245, 247
  tty, 245, 247
  UNIX, 248
ibufcount() (audio device method), 458
id()
  in module , 23
  TestCase method, 117
idcok() (method), 172
IDEA
  cipher, 407
ident (data in cd), 442
identchars (Cmd attribute), 135
idok() (method), 172
IEEE-754, 39
if
  statement, 3
ignorableWhitespace() (ContentHandler method), 382
ignore() (Stats method), 269
IGNORECASE (data in re), 84
ihave() (NNTPDataError method), 306
ihooks (standard module), 20
imageop (built-in module), 394
IMAP4
  protocol, 300
IMAP4 (class in imaplib), 300
IMAP4.abort (exception in imaplib), 300
IMAP4.error (exception in imaplib), 300
IMAP4.readonly (exception in imaplib), 300
imaplib (standard module), 300
imgfile (built-in module), 453
imghdr (standard module), 404
immedok() (method), 172
imp (built-in module), 20, 64
import
  statement, 20, 64
ImportError (exception in exceptions), 18
  in
  operator, 4, 7
INADDR_* (data in socket), 210
inch() (method), 172
Incomplete (exception in binascii), 340
IncrementalParser (class in xml.sax.xmlreader), 384
Independent JPEG Group, 454
index()
  array method, 128
  in module string, 78
  string method, 7
index (data in cd), 442
index() (list method), 11
IndexError (exception in exceptions), 18
indexOf() (in module operator), 46
IndexSizeErr (exception in xml.dom), 371
inet_aton() (in module socket), 211
inet_ntoa() (in module socket), 211
infile (shlex attribute), 137
Infinity, 23, 78
info() (NullTranslations method), 201
infolist() (ZipFile method), 233
InfoSeek Corporation, 263
ini file, 129
init()
  in module fm, 450
  in module mimetypes, 345
init_builtin() (in module imp), 66
init_color() (in module curses), 167
init_frozen() (in module imp), 66
init_pair() (in module curses), 167
inited (data in mimetypes), 345
initscr() (in module curses), 167
input()
  built-in function, 33
  in module , 23
  in module fileinput, 131
InputSource (class in xml.sax.xmlreader), 384
InputType (data in cStringIO), 95
insch() (method), 172
insdellin() (method), 172
insert()
array method, 128
list method, 11
insert_text() (in module readline), 235
insertBefore() (Node method), 366
insertln() (method), 173
insnstr() (method), 173
insort() (in module bisect), 126
insort_left() (in module bisect), 126
insort_right() (in module bisect), 126
inspect (standard module), 48
-insnstr() (method), 173
install()
in module gettext, 200
NullTranslations method, 201
install_opener() (in module urllib2), 287
instance() (in module new), 73
instancemethod() (in module new), 73
InstanceType (data in types), 42
istr() (method), 173
instream (shlex attribute), 137
int()
built-in function, 5
in module , 23
Int2AP() (in module imaplib), 301
integer
arbitrary precision, 409
division, 5
division, long, 5
literals, 5
literals, long, 5
object, 5
types, operations on, 6
Intel/DVI ADPCM, 391
interact()
in module code, 67
InteractiveConsole method, 68
Telnet method, 311
InteractiveConsole (class in code), 67
InteractiveInterpreter (class in code), 67
intern() (in module ), 24
internal_attr (ZipInfo attribute), 234
Internaldate2tuple() (in module imaplib), 301
internalSubset (DocumentType attribute), 367
Internet, 275
Internet Config, 284
InterpolationDepthError (exception in ConfigParser), 129
InterpolationError (exception in ConfigParser), 129
interpreter prompts, 32
intro (Cmd attribute), 135
IntType (data in types), 42
InuseAttributeErr (exception in xml.dom), 371
inv() (module operator), 45
InvalidAccessErr (exception in xml.dom), 371
InvalidCharacterErr (exception in xml.dom), 371
InvalidModificationErr (exception in xml.dom), 371
InvalidStateErr (exception in xml.dom), 371
invert() (module operator), 45
IOCTL (standard module), 248
ioctl() (module fcntl), 248
IOError (exception in exceptions), 18
IP_*(data in socket), 210
IPPORT_*(data in socket), 210
IPPROTO_*(data in socket), 210
IRIS Font Manager, 450
IRIX
threads, 217
is
operator, 4
is not
operator, 4
isbuiltin() (module imp), 66
is_data() (MultiFile method), 337
is_frozen() (module imp), 66
is_linetouched() (method), 173
is_wintouched() (method), 173
is_zipfile() (module zipfile), 232
isabs() (module os.path), 151
isAlive() (Thread method), 223
isalnum()
in module curses.ascii, 181
string method, 7
isalpha()
in module curses.ascii, 181
string method, 7
isascii() (module curses.ascii), 182
isatty()
Chunk method, 402
file method, 14
in module os, 143
isblank() (module curses.ascii), 182
isbuiltin() (module inspect), 50
isCallable() (module operator), 46
iscallable() (module inspect), 50
isctr1() (module curses.ascii), 182
iscode() (module inspect), 50
iscomment() (AddressList method), 332
isctrl() (module curses.ascii), 182
isDaemon() (Thread method), 223
isdigit()
in module curses.ascii, 182
string method, 7
isdir() (in module os.path), 151
isenabled() (in module gc), 34
isendwin() (in module curses), 167
ISEOF() (in module token), 427
isexpr()
   AST method, 420
   in module parser, 419
isfile() (in module os.path), 151
isfirstline() (in module fileinput), 132
isframe() (in module inspect), 50
isfunction() (in module inspect), 50
isgraph() (in module curses.ascii), 182
isheader() (AddressList method), 332
isinstance() (in module ), 24
iskeyword() (in module keyword), 427
islast() (AddressList method), 332
isleap() (in module calendar), 133
islink() (in module os.path), 151
islower()
   in module curses.ascii, 182
   string method, 7
isMappingType() (in module operator), 47
ismeta() (in module curses.ascii), 182
ismethod() (in module inspect), 50
ismodule() (in module inspect), 50
ismount() (in module os.path), 151
ISNONTERMINAL () (in module token), 427
isNumberType() (in module operator), 47
isprint() (in module curses.ascii), 182
ispunct() (in module curses.ascii), 182
isqueued() (in module fl), 446
isreadable()
   in module pprint, 70
   PrettyPrinter method, 71
isrecursive()
   in module pprint, 71
   PrettyPrinter method, 71
isReservedKey () (Morsel method), 321
isroutine() (in module inspect), 50
isSameNode() (Node method), 366
isSequenceType() (in module operator), 47
isSet() (Event method), 221
isspace()
   in module curses.ascii, 182
   string method, 7
isstdin() (in module fileinput), 132
issubclass() (in module ), 24
issuite()
   AST method, 420
   in module parser, 419
ISTERMINAL () (in module token), 427
istitle() (string method), 8
istraceback() (in module inspect), 50
isupper() in module curses.ascii, 182
   string method, 8
isdigit() (in module curses.ascii), 182
item()
   NamedNodeMap method, 370
   NodeList method, 367
items() (dictionary method), 11
itemsize (array attribute), 127
J
Jansen, Jack, 339
JFIF, 454
join()
   in module os.path, 151
   in module string, 79
   string method, 8
   Thread method, 222
joinfields() (in module string), 79
jpeg (built-in module), 454
js_output ()
   BaseCookie method, 320
   Morsel method, 321
jumpahead() (in module random), 123
K
kbhit() (in module msvcrt), 460
KDEDIR, 276
key (Morsel attribute), 321
KeyboardInterrupt (exception in exceptions), 18
KeyError (exception in exceptions), 18
keyname() (in module curses), 167
keypress() (method), 173
keys()
   method, 228
dictionary method, 11
keyword (standard module), 427
kill() (in module os), 147
killchar() (in module curses), 167
knee (standard module), 67
knownfiles (data in mimetypes), 345
Kuchling, Andrew, 407
L
L (data in re), 84
LambdaType (data in types), 42
LANG, 195, 196, 199, 200
LANGUAGE, 199, 200
language
   ABC, 4
   C, 4, 5
large files, 240
last()
   method, 229
dbhash method, 227
NNTPDataError method, 305
last (MultiFile attribute), 337
last_traceback (data in sys), 32
last_type (data in sys), 32
last_value (data in sys), 32
lastChild (Node attribute), 366
lastcmd (Cmd attribute), 135
lastgroup (MatchObject attribute), 88
lastindex (MatchObject attribute), 88
lastpart() (MimeWriter method), 336
LC_ALL, 199, 200
LC_ALL (data in locale), 197
LC_COLLATE (data in locale), 197
LC_CTYPE (data in locale), 197
LC_MESSAGES, 199, 200
LC_MESSAGES (data in locale), 197
LC_MONETARY (data in locale), 197
LC_NUMERIC (data in locale), 197
LC_TIME (data in locale), 197
ldexp() (in module math), 120
leapdays() (in module calendar), 120
leaveok() (method), 173
left_list (dircmp attribute), 157
left_only (dircmp attribute), 157
len()
  built-in function, 7, 11
  in module, 24
length
  NamedNodeMap attribute, 370
  NodeList attribute, 367
letters (data in string), 77
level (MultiFile attribute), 337
library (data in dbm), 244
light-weight processes, 216
lin2adpcm() (in module audioop), 392
lin2adpcm3() (in module audioop), 392
lin2lin() (in module audioop), 392
lin2ulaw() (in module audioop), 392
line-buffered I/O, 25
linecache (standard module), 53
linenol() (in module fileinput), 131
lineno
  class descriptor attribute, 429
  ExpatError attribute, 361
  shlex attribute, 137
LINES, 170
linesep (data in os), 150
link() (in module os), 145
list
  object, 6, 10
  type, operations on, 11
list()
  IMAP4 method, 302
  in module, 24
  NNTPDataError method, 305
  POP3 method, 299
listallfolders() (MH method), 348
listallsubfolders() (MH method), 348
listdir()
  in module dircache, 152
  in module os, 145
listen()
  dispatcher method, 324
  socket method, 213
listfolders() (MH method), 348
listmessages() (Folder method), 348
lists_folders() (MH method), 348
ListType (data in types), 42
literals
  complex number, 5
  floating point, 5
  hexadecimal, 5
  integer, 5
  long integer, 5
  numeric, 5
  octal, 5
ljust()
  in module string, 79
  string method, 8
LK_LOCK (data in msvcr), 459
LK_NBLCK (data in msvcr), 459
LK_NBRLOCK (data in msvcr), 459
LK_RLOCK (data in msvcr), 459
LK_UNLOCK (data in msvcr), 459
LNAME, 164
load()
  BaseCookie method, 320
  in module marshal, 62
  in module pickle, 56
load_compiled() (in module imp), 66
load_dynamic() (in module imp), 66
load_module() (in module imp), 65
load_source() (in module imp), 66
loads()
  in module marshal, 62
  in module pickle, 57
loadTestsFromModule() (TestLoader method), 118
loadTestsFromName() (TestLoader method), 118
loadTestsFromNames() (TestLoader method), 119
loadTestsFromTestCase() (TestLoader method), 118
LOCALE (data in re), 84
locale (standard module), 195
localeconv() (in module locale), 195
localName
   Attr attribute, 369
   Node attribute, 366
locals() (in module ), 24
localtime() (in module time), 161
Locator (class in xml.sax.xmlreader), 384
Lock () (in module threading), 217
lock()
   in module posixfile, 250
   mutex method, 164
locked() (lock method), 216
lockf()
   in module fcntl, 248
   in module fcntl, 250
locking() (in module msvcrt), 459
LockType (data in thread), 216
log()
   in module cmath, 121
   in module math, 120
log10()
   in module cmath, 121
   in module math, 120
log_data_time_string() (BaseHTTPRe-
   questHandler method), 317
log_error() (BaseHTTPRequestHandler
   method), 317
log_message() (BaseHTTPRequestHandler
   method), 317
log_request() (BaseHTTPRequestHandler
   method), 317
login()
   FTP method, 296
   IMAP4 method, 302
LOGNAME, 164, 296
lognormvariate() (in module random), 124
logout () (IMAP4 method), 302
long
   integer division, 5
   integer literals, 5
long()
   built-in function, 5, 78
   in module , 24
long integer
   object, 5
longimagedata () (in module rgbimg), 403
longname () (in module curses), 168
longstointag () (in module rgbimg), 403
LongType (data in types), 42
lookup()
   in module codecs, 96
   in module unicodedata, 100
LookupError (exception in exceptions), 17
lower()
   in module string, 79
   string method, 8
lowercase (data in string), 77
lseek () (in module os), 143
lshift () (in module operator), 45
lstat () (in module os), 145
lstrip()
   in module string, 79
   string method, 8
lsub() (IMAP4 method), 302
Lundh, Fredrik, 454
M
M (data in re), 84
macros (netrc attribute), 351
mailbox (standard module), 331, 346
mailcap (standard module), 343
Maildir (class in mailbox), 347
main() (in module unittest), 115
make_form() (in module fl), 445
make_parser() (in module xml.sax), 377
makedirs() (in module os), 145
makefile() (socket method), 213
makefolder() (MH method), 348
maketrans() (in module string), 79
map () (in module ), 24
mapcolor() (in module fl), 446
mapping
   object, 11
   types, operations on, 11
maps() (in module nis), 255
marshal (built-in module), 54, 55, 61
marshalling
   objects, 54
   masking
   operations, 6
match()
   in module nis, 254
   in module re, 85
   RegexObject method, 86
math (built-in module), 5, 119, 122
max()
   built-in function, 7
   in module , 24
   in module audioop, 392
MAX_INTERPOLATION_DEPTH (data in Config-
   Parser), 130
maxdict (Repr attribute), 72
maxint (data in sys), 32
MAXLEN (data in mimify), 350
maxlevel (Repr attribute), 72
maxlist (Repr attribute), 72
maxother (Repr attribute), 72
maxpp () (in module audioop), 392
maxstring (Repr attribute), 72
maxtuple (Repr attribute), 72
md5() (in module md5), 407
md5 (built-in module), 407
MemoryError (exception in exceptions), 18
Message
class in mhlib, 348
class in mimetools, 334
class in rfc822, 331
in module mimetools, 316
message digest, MD5, 407
MessageClass (BaseHTTPRequestHandler attribute), 316
meta() (in module curses), 168
method
object, 13
methods (class descriptor attribute), 429
MethodType (data in types), 42
MH (class in mhlib), 347
mHMailbox (class in mailbox), 347
MIME
base64 encoding, 345
content type, 344
headers, 277, 345
quoted-printable encoding, 346
mime_encode_header() (in module mimify), 350
mimeticlass (BaseHTTPRequestHandler attribute), 316
module
search path, 32, 54, 73
module() (in module new), 73
module (class descriptor attribute), 429
modules (data in sys), 32
ModuleType (data in types), 42
mono2grey() (in module imageop), 394
month() (in module calendar), 134
monthcalendar() (in module calendar), 133
monthrange() (in module calendar), 133
Morsel (class in Cookie), 320
mouseinterval() (in module curses), 168
mousemask() (in module curses), 168
move() (method), 173, 183, 224
movemessage() (Folder method), 349
MP, GNU library, 409
mpz() (in module mpz), 409
mpz (built-in module), 409
MPZType (data in mpz), 409
mstoblock() (CD player method), 443
msftoframe() (in module cd), 442
msg() (Telnet method), 310
MSG_* (data in socket), 210
msvcrt (built-in module), 459
mt_interact() (Telnet method), 311
mtime() (RobotFileParser method), 351
mul()
in module audioop, 393
in module operator, 45
MultiFile (class in multifile), 336
multifile (standard module), 336
MULTILINE (data in re), 84
mutable
sequence types, 10
sequence types, operations on, 11
MutableString (class in UserString), 44
mutex
class in mutex, 164
standard module, 164
mvderwin() (method), 173
mvwin() (method), 173

N
name() (in module unicodedata), 100
name
Attr attribute, 369
class descriptor attribute, 429
data in os, 140
DocumentType attribute, 367
file attribute, 16
Index 501
OpenKeyEx() (in module _winreg), 462
openlog() (in module syslog), 255
openmessage() (Message method), 349
openport() (in module al), 439
openpty() in module os, 143
openpty() in module pty, 247
operation
concatenation, 7
repetition, 7
slice, 7
subscription, 7
operation
bit-string, 6
Boolean, 3, 4
masking, 6
shifting, 6
operations on
dictionary type, 11
integer types, 6
list type, 11
mapping types, 11
mutable sequence types, 11
numeric types, 5
sequence types, 7, 11
operator
==, 4
and, 3, 4
comparison, 4
in, 4, 7
is, 4
is not, 4
not, 4
not in, 4, 7
or, 3, 4
operator (built-in module), 44
opname (data in dis), 431
options() (ConfigParser method), 130
optionxform() (ConfigParser method), 131
or
operator, 3, 4
or__() (in module operator), 45
ord() (in module), 25
ordered_attributes (xmlparser attribute), 358
os (standard module), 14, 33, 139, 239
os.path (standard module), 150
OSError (exception in exceptions), 18
output()
BaseCookie method, 320
Morsel method, 321
OutputString() (Morsel method), 321
OutputType (data in cStringIO), 95
OverflowError (exception in exceptions), 18
overlay() (method), 174
Overmars, Mark, 445
overwrite() (method), 174

P
P_DETACH (data in os), 148
P_NOWAIT (data in os), 148
P_NOWAITO (data in os), 148
P_OVERLAY (data in os), 148
P_WAIT (data in os), 148
pack() (in module struct), 88
pack_array() (Packer method), 342
pack_bytes() (Packer method), 342
pack_double() (Packer method), 341
pack_farray() (Packer method), 342
pack_float() (Packer method), 341
pack_fopaque() (Packer method), 341
pack_fstring() (Packer method), 341
pack_list() (Packer method), 342
packOpaque() (Packer method), 342
pack_string() (Packer method), 341
package, 74
Packer (class in xdrlib), 341
packing
binary data, 88
PAGER, 259
pair_content() (in module curses), 169
pair_number() (in module curses), 169
pardir (data in os), 150
parent (BaseHandler attribute), 290
parentNode (Node attribute), 365
paretovariate() (in module random), 124
Parse() (xmlparser method), 358
parse()
in module cgi, 279
in module xml.dom.minidom, 373
in module xml.dom.pulldom, 377
in module xml.sax, 377
RobotFileParser method, 351
XMLReader method, 384
parse_and_bind() (in module readline), 235
parse_header() (in module cgi), 280
parse_multipart() (in module cgi), 280
parse_qs() (in module cgi), 279
parsedate() (in module rfc822), 331
parsedate_tz() (in module rfc822), 331
ParseFile() (xmlparser method), 358
ParseFlags() (in module imaplib), 301
parseframe() (CD parser method), 444
parser (built-in module), 417
ParserCreate() (in module xml.parsers.expat), 357
ParserError (exception in parser), 420
parsesequence() (Folder method), 349
parseString()
in module xml.dom.minidom, 373
in module xml.dom.pulldom, 377
in module xml.sax, 378
parsing
  Python source code, 417
  URL, 312
ParsingError (exception in ConfigParser), 130
partial() (IMAP4 method), 302
pass_() (POP3 method), 299
PATH, 150, 281, 283
path
  configuration file, 74
  module search, 32, 54, 73
  operations, 150
path
  BaseHTTPRequestHandler attribute, 316
  data in os, 140
  data in sys, 32
pathconf() (in module os), 145
pathconf_names (data in os), 145
pathsep (data in os), 150
pattern (RegexObject attribute), 87
pause() (in module signal), 208
PAUSED (data in cd), 442
Pdb (class in pdb), 257
pdb (standard module), 32, 257
persistence, 54
persistent
  objects, 54
pformat()
in module pprint, 70
  PrettyPrinter method, 71
PGP, 407
pi
  data in cmath, 121
  data in math, 120
pick() (in module gl), 452
pickle() (in module copy_reg), 59
pickle (standard module), 54, 59, 61
Pickler (class in pickle), 56
pickling
  objects, 54
PicklingError (exception in pickle), 57
pid (Popen4 attribute), 159
PIL (the Python Imaging Library), 454
pipe() (in module os), 143
pipes (standard module), 249
PKG_DIRECTORY (data in imp), 65
platform (data in sys), 32
play() (CD player method), 443
playabs() (CD player method), 443
plock() (in module os), 148
pm() (in module db), 258
pnum (data in cd), 442
poll()
  method, 215
  in module select, 214
  Popen4 method, 159
pop()
  array method, 128
  list method, 11
  MultiFile method, 337
POP3
  protocol, 298
POP3 (class in poplib), 299
pop_alignment() (formatter method), 328
pop_font() (formatter method), 329
pop_margin() (formatter method), 329
pop_source() (shlex method), 136
pop_style() (formatter method), 329
popen()
  in module os, 142
  in module os, 215
popen2()
  in module os, 142
  in module popen2, 158
popen2 (standard module), 158
Popen3 (class in popen2), 158
popen3()
  in module os, 142
  in module popen2, 158
Popen4 (class in popen2), 158
popen4()
  in module os, 142
  in module popen2, 158
poplib (standard module), 298
PortableUnixMailbox (class in mailbox), 347
pos() (in module operator), 45
pos (MatchObject attribute), 88
posix (built-in module), 239
posixfile (built-in module), 250
POSIX
  file object, 250
  I/O control, 245, 247
  threads, 216
post() (NNTPDataError method), 306
post_mortem() (in module pdb), 258
postcmd() (Cmd method), 135
postloop() (Cmd method), 135
pow()
  in module , 25
  in module math, 120
powm() (in module mp), 409
pprint()
  in module pprint, 70
  PrettyPrinter method, 71
pprint (standard module), 69
prcal() (in module calendar), 134
pre (standard module), 80
precmd () (Cmd method), 135
prefix
  Attr attribute, 369
  data in sys, 32
  Node attribute, 366
preloop () (Cmd method), 135
prepare_input_source() (in module xml.sax.saxutils), 383
prepend () (Template method), 250
Pretty Good Privacy, 407
PrettyPrinter (class in pprint), 69
preventremoval() (CD player method), 443
previous()
  method, 229
  dbhash method, 227
previousSibling (Node attribute), 366
print
  statement, 3
print_callees() (Stats method), 269
print_callers() (Stats method), 268
print_directory() (in module cgi), 280
print_environ() (in module cgi), 280
print_environment_usage() (in module cgi), 280
print_exc() (in module traceback), 52
print_exception() (in module traceback), 52
print_form() (in module cgi), 280
print_last () (in module traceback), 52
print_stack () (in module traceback), 52
print_stats () (Stats method), 268
print_tb() (in module traceback), 52
printable (data in string), 77
printdir() (ZipFile method), 233
printf-style formatting, 9
prmonth() (in module calendar), 134
process
  group, 140, 141
  id, 141
  id of parent, 141
  killing, 147
  signalling, 147
process_request () (in module SocketServer), 314
processes, light-weight, 216
processingInstruction() (ContentHandler method), 382
ProcessingInstructionHandler() (xml-parser method), 360
profile (standard module), 266
profile function, 33
profiler, 33
profiling, deterministic, 263
prompt (Cmd attribute), 135
prompt_user_passwd() (FancyURLopener method), 286
prompts, interpreter, 32
property_declaration_handler (data in xml.sax.handler), 380
property_dom_node (data in xml.sax.handler), 380
property_lexical_handler (data in xml.sax.handler), 380
property_xml_string (data in xml.sax.handler), 380
protocol
  CGI, 277
  FTP, 285, 295
  Gopher, 285, 286, 298
  HTTP, 277, 285, 293, 315
  IMAP4, 300
  NNTP, 303
  POP3, 298
  SMTP, 306
PROTOCOL_VERSION (IMAP4 attribute), 303
protocol_version (BaseHTTPRequestHandler attribute), 316
proxy () (in module weakref), 36
ProxyBasicAuthHandler (class in urllib2), 288
ProxyDigestAuthHandler (class in urllib2), 289
ProxyHandler (class in urllib2), 288
ProxyType (data in weakref), 36
ProxyTypes (data in weakref), 36
prstr () (in module fm), 450
ps1 (data in sys), 32
ps2 (data in sys), 32
pstats (standard module), 267
pthreads, 216
ptime (data in cd), 442
pty (standard module), 143, 247
publicId (DocumentType attribute), 367
FullDOM (class in xml.dom.pulldom), 377
punctuation (data in string), 77
push()
  InteractiveConsole method, 69
  MultiFile method, 337
push_alignment() (formatter method), 328
push_font() (formatter method), 329
push_margin() (formatter method), 329
push_source() (shlex method), 136
push_style() (formatter method), 329
push_token() (shlex method), 136
put () (Queue method), 223
put_nowait() (Queue method), 223
putch() (in module msvcrtd), 460
putenv() (in module os), 141
putheader() (HTTP method), 294
putsequences() (Folder method), 349
putwin() (method), 174
pwd() (FTP method), 298
pwd (built-in module), 151, 240
pwlcurve() (in module gl), 452
py_compile (standard module), 429
PY_COMPILLED (data in imp), 65
PY_FROZEN (data in imp), 65
PY_RESOURCE (data in imp), 65
PY_SOURCE (data in imp), 65
pycclbr (standard module), 428
pyexpat (built-in module), 357
PyOpenGL, 453
Python Enhancement Proposals
  PEP 0205, 36
Python Imaging Library, 454
PYTHON, 364
PYTHONPATH, 32, 281, 468
PYTHONSTARTUP, 74, 236
PYTHON2K, 159, 160
PyZipFile (class in zipfile), 232

Q
qdevice() (in module fl), 446
qenter() (in module fl), 446
qiflush() (in module curses), 169
qreset() (in module fl), 446
qsize() (Queue method), 223
qtest() (in module fl), 446
QueryInfoKey() (in module _winreg), 462
queryparams() (in module al), 439
QueryValue() (in module _winreg), 462
QueryValueEx() (in module _winreg), 463
Queue
class in Queue, 223
standard module, 223
quick_ratio() (SequenceMatcher method), 93
quit()
  FTP method, 298
  NNTPDataError method, 306
  POP3 method, 299
  SMTP method, 309
quopri (standard module), 346
quote() (in module urllib), 284
quote_plus() (in module urllib), 284
quoted-printable
  encoding, 346
quotes (shlex attribute), 137

R
r_eval() (RExec method), 415
r_exec() (RExec method), 415
r_execfile() (RExec method), 415
r_import() (RExec method), 415
R_OK (data in os), 144
r_open() (RExec method), 415
r_reload() (RExec method), 415
r_unload() (RExec method), 415
raise
  statement, 17
randint()
    in module random, 123
    in module wrandom, 125
random()
    in module random, 124
    in module wrandom, 125
random (standard module), 122
randrange() (in module random), 123
range() (in module), 25
Rat (demo module), 409
ratecv() (in module audioop), 393
ratio() (SequenceMatcher method), 93
rational numbers, 409
raw() (in module curses), 169
raw_input()
  built-in function, 33
    in module, 26
    InteractiveConsole method, 69
re
  MatchObject attribute, 88
  standard module, 10, 77, 80, 193
read()
  method, 225
  array method, 128
  audio device method, 458
  Chunk method, 402
  ConfigParser method, 130
  file method, 15
  in module imgfile, 454
  in module os, 143
  MultiFile method, 336
  RobotFileParser method, 351
  StreamReader method, 99
  ZipFile method, 233
read_all() (Telnet method), 310
read_byte() (method), 225
read_eager() (Telnet method), 310
read_history_file() (in module readline), 235
read_init_file() (in module readline), 235
read_lazy() (Telnet method), 310
read_mime_types() (in module mimetypes), 345
read_some() (Telnet method), 310
read_token() (shlex method), 136
read_until() (Telnet method), 310
read_very_eager() (Telnet method), 310
read_veryLazy() (Telnet method), 310
readable() (dispatcher method), 324
readadda() (CD player method), 443
readfp() (ConfigParser method), 130
readframes()
  aifc method, 396
  AU_read method, 398
  Wave_read method, 400
readline()
  method, 225
  file method, 15
  MultiFile method, 336
  StreamReader method, 99
readline (built-in module), 235
readlines()
  file method, 15
  MultiFile method, 336
  StreamReader method, 99
readlink() (in module os), 145
readmodule() (in module pycbr), 429
readsamps() (audio port method), 440
readscaled() (in module imgfile), 454
READY (data in cd), 442
Real Media File Format, 401
real_quick_ratio() (SequenceMatcher method), 93
recent() (IMAP4 method), 302
rectangle() (in module curses.textpad), 179
recv()
  dispatcher method, 324
  socket method, 213
recvfrom() (socket method), 213
redraw_form() (form method), 446
redraw_object() (FORMS object method), 449
redrawln() (method), 174
redrawwin() (method), 174
reduce() (in module ), 26
ref() (in module weakref), 35
ReferenceError (exception in weakref), 36
ReferenceType (data in weakref), 36
removemessages() (Folder method), 349
refresh() (method), 174
register()
  method, 215
  in module atexit, 40
  in module codecs, 96
  in module webbrowser, 276
registerDOMImplementation() (in module xml.dom), 364
RegLoadKey() (in module _winreg), 462
relative
  URL, 312
release()
  Condition method, 220
  lock method, 216
  Semaphore method, 221
  Thread method, 218, 219
reload()
  built-in function, 32, 65, 67
  in module , 26
remove()
  array method, 128
  in module os, 145
  list method, 11
remove_option() (ConfigParser method), 131
remove_section() (ConfigParser method), 131
rename() (Element method), 369
renameAttribute() (Element method), 369
renameAttributeNS() (Element method), 369
removecallback() (CD parser method), 444
removeChild() (Node method), 366
renamedirs() (in module os), 145
removeUnbalanced() (Folder method), 349
rename()
  FTP method, 297
  IMAP4 method, 302
  in module os, 146
renames() (in module os), 146
reorganize() (in module gdbm), 245
repeat() (in module operator), 46
replication
  operation, 7
replace()
  method, 184
  in module string, 80
  string method, 8
replaceChild() (Node method), 367
report() (dircmp method), 157
report_full_closure() (dircmp method), 157
report_partial_closure() (dircmp method), 157
report_unbalanced() (SGMLParser method), 355
Repr (class in repr), 71
repr()
  in module , 27
  in module repr, 72
  Repr method, 72
repr (standard module), 71
repr1() (Repr method), 72
Request (class in urllib2), 288
request_queue_size (data in SocketServer), 314
request_version (BaseHTTPRequestHandler attribute), 316
RequestHandlerClass (data in SocketServer), 314
reserved (ZipInfo attribute), 234
reset()
   DOMEventStream method, 377
   in module statcache, 155
   IncrementalParser method, 385
   Packer method, 341
   SGMLParser method, 353
   StreamReader method, 99
   StreamWriter method, 98
   Template method, 249
   Unpacker method, 342
   XMLParser method, 388
reset ()
   prog (in module curses), 169
reset ()
   shell (in module curses), 169
resetbuffer () (InteractiveConsole method), 169
resetlocale() (in module locale), 197
resetparser () (CD parser method), 444
resetwarnings () (in module warnings), 64
resize () (method), 225
resolveEntity () (EntityResolver method), 382
resource (built-in module), 252
response () (IMAP4 method), 382
responses (BaseHTTPRequestHandler attribute), 316
retr () (POP3 method), 299
retrbinary () (FTP method), 296
retrlines () (URLopener method), 286
retrlines () (FTP method), 297
returns_unicode (xmlparser attribute), 359
reverse ()
   array method, 128
   in module audioop, 393
   list method, 11
reverse_order () (Stats method), 268
rewind ()
   aifc method, 396
   AU_read method, 398
   Wave_read method, 400
rewindBody () (AddressList method), 332
RExec (class in rexec), 414
rexml (standard module), 20, 414
RFC
RFC 1014, 341
RFC 1321, 407
RFC 1521, 345, 346
RFC 1524, 344
RFC 1725, 298
RFC 1730, 300
RFC 1738, 313
RFC 1766, 196
RFC 1808, 313
RFC 1832, 341
RFC 1866, 355, 356
RFC 1869, 306, 307
RFC 2060, 300
RFC 2068, 319
RFC 2109, 319, 320
RFC 2396, 313
RFC 821, 306, 307
RFC 822, 129, 162, 201, 294, 308, 309, 331, 332
RFC 854, 309, 310
RFC 959, 295
RFC 977, 303
rfc822 (standard module), 331, 334
rfile (BaseHTTPRequestHandler attribute), 316
rfind ()
   in module string, 78
   string method, 8
rgb_to_hls () (in module colormapsys), 403
rgb_to_hsv () (in module colormapsys), 403
rgb_to_yiq () (in module colormapsys), 403
rgbimg (built-in module), 403
right_list (dir attr), 157
right_only (dir attr), 157
rindex ()
   in module string, 79
   string method, 8
rjust ()
   in module string, 79
   string method, 8
rlcompleter (standard module), 236
rlecode_hqx () (in module binascii), 340
rledecode_hqx () (in module binascii), 340
RLIMIT AS (data in resource), 253
RLIMIT_CORE (data in resource), 252
RLIMIT_CPU (data in resource), 253
RLIMIT_DATA (data in resource), 253
RLIMIT_FSIZE (data in resource), 253
RLIMIT_MEMLOC (data in resource), 253
RLIMIT_NOFILE (data in resource), 253
RLIMIT_NPROC (data in resource), 253
RLIMIT_OFILE (data in resource), 253
RLIMIT_RSS (data in resource), 253
RLIMIT_STACK (data in resource), 253
RLIMIT_VMEM (data in resource), 253
RLock () (in module threading), 217
rmdir () (FTP method), 298
rmdir () (in module os), 146
RMFF, 401
rms () (in module audioop), 393
rmtree () (in module shutil), 194
rnopen () (in module bsddb), 228
RobotFileParser (class in robotparser), 351
robotparser (standard module), 351

508 Index
Index

robots.txt, 351
rotor (built-in module), 410
round() (in module), 27
rpop() (POP3 method), 299
rset() (POP3 method), 299
rshift() (in module operator), 45
rstrip() in module string, 79
ruler (Cmd attribute), 135
run() in module pdb, 258
    in module profile, 266
    scheduler method, 163
    TestCase method, 116
    Thread method, 222
runcall() (in module pdb), 258
runcode() (InteractiveConsole method), 68
runsource() (InteractiveConsole method), 68
RuntimeError (exception in exceptions), 19
RuntimeWarning (exception in exceptions), 20
RUSAGE_BOTH (data in resource), 254
RUSAGE_CHILDREN (data in resource), 254
RUSAGE_SELF (data in resource), 254
S
    S (data in re), 84
    s_eval() (RExec method), 415
    s_exec() (RExec method), 415
    s_execfile() (RExec method), 415
    S_IFMT() (in module stat), 153
    S_IMODE() (in module stat), 153
    s_import() (RExec method), 415
    S_ISBLK() (in module stat), 153
    S_ISCHR() (in module stat), 153
    S_ISDIR() (in module stat), 153
    S_ISFIFO() (in module stat), 153
    S_ISLNK() (in module stat), 153
    S_ISREG() (in module stat), 153
    S_ISSOCK() (in module stat), 153
    s_reload() (RExec method), 415
    s_unload() (RExec method), 415
    saferepr() (in module pprint), 71
    same_files (dircmp attribute), 158
    samefile() (in module os.path), 151
    sameopenfile() (in module os.path), 152
    samestat() (in module os.stat), 152
    save_bgn() (HTMLParser method), 357
    save_end() (HTMLParser method), 357
    SaveKey() (in module _winreg), 463
    SAX2DOM (class in xml.dom.pulldom), 377
    SAXException (exception in xml.sax), 378
    SAXNotRecognizedException (exception in xml.sax), 378
    SAXNotSupportedException (exception in xml.sax), 378
    SAXParseException (exception in xml.sax), 378
    scale() (in module imageop), 394
    scalefont() (in module fm), 450
    sched (standard module), 162
    scheduler (class in sched), 162
    sci() (in module fppformat), 94
    scroll() (method), 174
    scrollok() (method), 174
    search
        path, module, 32, 54, 73
        search()
            IMAP4 method, 302
            in module re, 85
            RegexObject method, 86
    SEARCH_ERROR (data in imp), 66
    section_divider() (MultiFile method), 337
    sections() (ConfigParser method), 130
    Secure Hash Algorithm, 408
    seed()
        in module random, 123
        in module whrandom, 125
        whrandom method, 125
    seek()
        method, 225
        CD player method, 443
        Chunk method, 402
        file method, 15
        MultiFile method, 336
    SEEK_CUR (data in posixfile), 250
    SEEK_END (data in posixfile), 250
    SEEK_SET (data in posixfile), 250
    seekblock() (CD player method), 443
    seektrack() (CD player method), 444
    select()
        IMAP4 method, 302
        in module gl, 452
        in module select, 214
    select (built-in module), 214
    Semaphore() (in module threading), 217
    Semaphore (class in threading), 220
    semaphores, binary, 216
    send()
        dispatcher method, 324
        HTTP method, 294
        socket method, 213
    send_error() (BaseHTTPRequestHandler method), 317
    send_flowing_data() (writer method), 330
send_header() (BaseHTTPRequestHandler method), 317
send_hor_rule() (writer method), 330
send_label_data() (writer method), 330
send_line_break() (writer method), 330
send_literal_data() (writer method), 330
send_line_break() (writer method), 330
send_label_data() (writer method), 330
send_query() (in module gopherlib), 298
send_response() (BaseHTTPRequestHandler method), 317
send_selector() (in module gopherlib), 298
sendcmd() (FTP method), 296
sendmail() (SMTP method), 308
sendto() (socket method), 213
sep (data in os), 150
sequence
  object, 6
  types, mutable, 10
  types, operations on, 7, 11
  types, operations on mutable, 11
sequence2ast() (in module parser), 418
sequenceIncludes() (in module operator), 46
SequenceMatcher (in module difflib), 91
SerialCookie (in module Cookie), 319
serializing
  objects, 54
serve_forever() (in module SocketServer), 314
server
  WWW, 277, 315
server_activate() (in module SocketServer), 314
server_address (data in SocketServer), 314
server_bind() (in module SocketServer), 315
server_version
  BaseHTTPRequestHandler attribute, 316
  SimpleHTTPRequestHandler attribute, 318
set()
  ConfigParser method, 130
  Event method, 221
  Morsel method, 321
set_call_back() (FORMS object method), 448
set_completer() (in module readline), 235
set_completer_delims() (in module readline), 236
set_debug() (in module gc), 34
set_debuglevel() (FTP method), 296
  HTTP method, 294
  NNTPDataError method, 305
  SMTP method, 307
  Telnet method, 311
set_event_call_back() (in module fl), 445
set_form_position() (form method), 446
set_graphics_mode() (in module fl), 445
set_history_length() (in module readline), 235
set_location() (method), 228
set_pasv() (FTP method), 297
set_position() (Unpacker method), 342
set_proxy() (Request method), 289
set_seq1() (SequenceMatcher method), 91
set_seq2() (SequenceMatcher method), 92
set_seqs() (SequenceMatcher method), 91
set_spacing() (formatter method), 329
set_threshold() (in module gc), 34
set_trace() (in module pdb), 258
set_url() (RobotFileParser method), 351
set_userptr() (method), 184
setattr() (in module ), 27
setAttribute() (Element method), 369
setAttributeNode() (Element method), 369
setAttributeNodeNS() (Element method), 369
setAttributeNS() (Element method), 369
SetBase() (xmlparser method), 358
setblocking() (socket method), 213
setBytes() (InputSource method), 386
setcbreak() (in module tty), 247
setchannels() (audio configuration method), 440
setCharacterStream() (InputSource method), 386
setcheckinterval() (in module sys), 33
setcomptype() (aifc method, 396
  AU_write method, 399
  Wave_write method, 401
setconfig() (audio port method), 441
setContentHandler() (XMLReader method), 384
setcontext() (MH method), 348
setCurrent() (Folder method), 349
setDaemon() (Thread method), 223
setdefaultencoding() (in module sys), 33
setDocumentLocator() (ContentHandler method), 380
setDTDHandler() (XMLReader method), 385
setegid() (in module os), 141
setEncoding() (InputSource method), 386
setEntityResolver() (XMLReader method), 385
setErrorHandler() (XMLReader method), 385
seteuid() (in module os), 141
setFeature() (XMLReader method), 385
setfillpoint() (audio port method), 441
setfirstweekday() (in module calendar), 133
setframe() (audio configuration method), 440
setframerate() (in module âm), 450
setframerate() (aifc method, 396
AU_write method, 399
Wave_write method, 401
setgid() (in module os), 141
setinfo() (audio device method), 458
setItem() (in module operator), 46
setkey() (rotor method), 410
setlast() (Folder method), 349
setliteral()
  SGMLParser method, 354
  XMLParser method, 388
setLocale() (XMLReader method), 385
setlocale() (in module locale), 195
setlogmask() (in module syslog), 255
setmark() (aifc method), 397
setMaxConns() (CacheFTPHandler method), 293
setmode() (in module msvcrt), 459
setName() (Thread method), 222
setnchannels()
  aifc method, 396
  AU_write method, 399
  Wave_write method, 401
setnframes()
  aifc method, 396
  AU_write method, 399
  Wave_write method, 401
setnomoretags()
  SGMLParser method, 353
  XMLParser method, 388
setoption() (in module jpeg), 454
setparams()
  aifc method, 396
  AU_write method, 399
  in module al, 440
  Wave_write method, 401
setpath() (in module fm), 450
setpgid() (in module os), 141
setpgrp() (in module os), 141
setpos()
  aifc method, 396
  AU_read method, 399
  Wave_read method, 401
setprofile() (in module sys), 33
setProperty() (XMLReader method), 385
setPublicId() (InputSource method), 386
setqueuesize() (audio configuration method), 440
setraw() (in module tty), 247
setrecursionlimit() (in module sys), 33
setregid() (in module os), 141
setreuid() (in module os), 141
setrlimit() (in module resource), 252
setsampfmt() (audio configuration method), 440
setsampwidth()
  aifc method, 396
  AU_write method, 399
  Wave_write method, 401
setstrrreg() (method), 174
setsysid() (in module os), 141
settimeouts() (socket method), 213
setState() (in module random), 123
setSystemId() (InputSource method), 386
setsyscall() (module curses), 169
setTimeOut() (CacheFTPHandler method), 293
settrace() (in module sys), 33
setuid() (in module os), 141
setUp() (TestCase method), 116
setup() (in module SocketServer), 315
setupterm() (module curses), 169
SetValue() (module _winreg), 463
SetValueEx() (module _winreg), 463
setwidth() (audio configuration method), 440
SGML, 353
sgmllib (standard module), 353, 355
SGMLParser
  class in sgmllib, 353
  in module sgmllib, 355
sha (built-in module), 408
shelve (standard module), 54, 59, 61
shifting
  operations, 6
shlex
  class in shlex, 136
  standard module, 136
shortDescription() (TestCase method), 117
short_description() (method), 184
show() (method), 445
show_choice() (module fl), 445
show_file_selector() (module fl), 446
show_form() (form method), 446
show_input() (module fl), 446
show_message() (module fl), 445
show_object() (FORMS object method), 449
show_question() (module fl), 445
showsyntaxerror() (InteractiveConsole method), 68
showtraceback() (InteractiveConsole method), 68
showwarning() (module warnings), 64
shuffle() (module random), 123
shutdown() (socket method), 213
shutil (standard module), 193
SIG* (data in signal), 208
SIG_DFL (data in signal), 208
SIG_IGN (data in signal), 208
signal() (module signal), 208
signal (built-in module), 207, 216
Simple Mail Transfer Protocol, 306
SimpleCookie (class in Cookie), 319
SimpleHTTPRequestHandler (class in Simple
HTTPServer), 318
SimpleHTTPServer (standard module), 315, 318
sin() in module cmath, 121
        in module math, 120
sinh() in module cmath, 121
        in module math, 120
site (standard module), 73
site-packages directory, 74
site-python directory, 74
sitecustomize (module), 74
size() method, 225
        FTP method, 298
sizeofImage() (in module rgbimg), 403
skip() (Chunk method), 402
skippedEntity() (ContentHandler method), 382
slave() (NNTPDataError method), 306
sleep() (in module time), 161
slice
        assignment, 11
        operation, 7
slice()
        built-in function, 43, 437
        in module , 27
SliceType (data in types), 43
SmartCookie (class in Cookie), 319
SMTP
        protocol, 306
SMTP (class in smtplib), 306
SMTPConnectError (exception in smtplib), 307
SMTPDataFrame (exception in smtplib), 307
SMTPException (exception in smtplib), 307
SMTPHeloError (exception in smtplib), 307
smtplib (standard module), 306
SMTPRecipientsRefused (exception in smtp
lib), 307
SMTPResponseException (exception in smtplib), 307
SMTPSenderRefused (exception in smtplib), 307
SMTPServerDisconnected (exception in smtp
lib), 307
SND_ALIAS (data in winsound), 465
SND_ASYNC (data in winsound), 466
SND_FILENAME (data in winsound), 465
SND_LOOP (data in winsound), 465
SND_MEMORY (data in winsound), 465
SND_NODEFAULT (data in winsound), 466
SND_NOWAIT (data in winsound), 466
SND_PURGE (data in winsound), 466
sndhdr (standard module), 404
SO_* (data in socket), 210
SOCK_DGRAM (data in socket), 210
SOCK_RAW (data in socket), 210
SOCK_RDM (data in socket), 210
SOCK_SEQPACKET (data in socket), 210
SOCK_STREAM (data in socket), 210
socket
        object, 209
socket()
        IMAP4 method, 303
        in module socket, 211
socket
        built-in module, 14, 209, 275
        data in SocketServer, 314
socket () (in module socket), 215
socket_type (data in SocketServer), 314
SocketServer (standard module), 313
SocketType (data in socket), 212
softspace (file attribute), 16
SOL_* (data in socket), 210
SOMAXCONN (data in socket), 210
sort () (list method), 11
sort_stats() (Stats method), 267
sortTestMethodsUsing (TestLoader attribute),
        119
source (shlex attribute), 137
sourcehook () (shlex method), 136
span () (MatchObject method), 88
spawn () (in module pty), 247
spawnv () (in module os), 148
spawnve () (in module os), 148
specified_attributes (xmlparser attribute),
        359
split()
        in module os.path, 152
        in module re, 85
        in module string, 79
                RegexObject method, 86
                string method, 8
splitdrive () (in module os.path), 152
splitext () (in module os.path), 152
splitfields () (in module string), 79
splitlines () (string method), 8
sprintf-style formatting, 9
sqrt()
        in module cmath, 121
        in module math, 120
        in module mpz, 409
sqrtrem () (in module mpz), 409
ST_ATIME (data in stat), 154
ST_CTIME (data in stat), 154
ST_DEV (data in stat), 154
ST_GID (data in stat), 154
ST_INO (data in stat), 154
ST_MODE (data in stat), 154
ST_MTIME (data in stat), 154
ST_NLINK (data in stat), 154
ST_SIZE (data in stat), 154
ST_UID (data in stat), 154
stack() (in module inspect), 51
stackable
  streams, 96
StandardError (exception in exceptions), 17
standend() (method), 174
standout() (method), 174
start()
  MatchObject method, 87
  Thread method, 222
start_color() (in module curses), 169
start_new_thread() (in module thread), 216
startbody() (MimeWriter method), 335
StartCdataSectionHandler() (xmlparser method), 360
StartDoctypeDeclHandler() (xmlparser method), 359
startDocument() (ContentHandler method), 381
startElement() (ContentHandler method), 381
StartElementHandler() (xmlparser method), 381
startElementNS() (xmlparser method), 381
startfile() (in module os), 148
startmultipartbody() (MimeWriter method), 336
StartNamespaceDeclHandler() (xmlparser method), 360
startPrefixMapping() (ContentHandler method), 381
startswith() (string method), 8
startTest() (TestResult method), 118
stat()
  in module os, 146
  in module statcache, 155
  NNTPDataError method, 305
  POP3 method, 299
stat(standard module), 146, 153
statcache(standard module), 155
statement
  assert, 17
  del, 11
  except, 17
  exec, 14
  if, 3
  import, 20, 64
  print, 3
  raise, 17
  try, 17
  while, 3
Stats (class in pstats), 267
status() (IMAP4 method), 303
statvfs() (in module os), 146
statvfs(standard module), 146, 156
stderr (data in sys), 33
stdin (data in sys), 33
stdout (data in sys), 33
STILL (data in cd), 442
stop()
  CD player method, 444
  TestResult method, 118
stopTest() (TestResult method), 118
storbinary() (FTP method), 297
store() (IMAP4 method), 303
storlines() (FTP method), 297
str()
  in module, 27
  in module locale, 197
strcoll() (in module locale), 197
StreamReader (class in codecs), 98
StreamReaderWriter (class in codecs), 99
StreamReorder (class in codecs), 100
streams, 96
  stackable, 96
StreamWriter (class in codecs), 98
strip() (in module os), 141
strftime() (in module time), 161
string
  documentation, 421
  formatting, 9
  object, 6
string
  MatchObject attribute, 88
  standard module, 10, 77, 197, 198
StringIO
  class in StringIO, 95
  standard module, 95
StringType (data in types), 42
strip()
  in module string, 79
  string method, 8
strip_dirs() (Stats method), 267
stripspaces (Textbox attribute), 180
strop (built-in module), 80, 198
strftime() (in module time), 162
struct (built-in module), 88, 213
structures
  C, 88
strxfrm() (in module locale), 197
sub()
  in module operator, 44
  in module re, 85
RegexObject method, 86
subdirs (dircmp attribute), 158
subn() in module re, 86
RegexObject method, 86
subpad() (method), 174
subscribe() (IMAP4 method), 303
subscript assignment, 11
operation, 7
subwin() (method), 175
suffix_map (data in mimetypes), 345
suite() (in module parser), 418
suiteClass (TestLoader attribute), 119
sunau (standard module), 397
SUNAUDIODEV (standard module), 457, 458
sunaudiodev (built-in module), 457, 458
super (class descriptor attribute), 429
swapcase()
in module string, 79
string method, 8
sym() (method), 243
sym_name (data in symbol), 426
symbol (standard module), 426
symbol table, 3
symlink () (in module os), 146
sync()
method, 229
dbhash method, 227
in module gdbm, 245
syncdown () (method), 175
syncok () (method), 175
syncup () (method), 175
syntax_error () (XMLParser method), 389
SyntaxErr (exception in xml.dom), 371
SyntaxError (exception in exceptions), 19
SyntaxWarning (exception in exceptions), 20
sys (built-in module), 29
sys_version (BaseHTTPRequestHandler attribute), 316
sysconf() (in module os), 149
sysconf_names (data in os), 150
syslog() (in module syslog), 255
syslog (built-in module), 255
system() (in module os), 148
SystemError (exception in exceptions), 19
SystemExit (exception in exceptions), 19
systemId (DocumentType attribute), 367

T
tabnanny (standard module), 428
tagName (Element attribute), 368
tan()
in module cmath, 121
in module math, 120
tanh()
in module cmath, 121
in module math, 120
target (ProcessingInstruction attribute), 370
tb_lineno() (in module traceback), 53
tcdrain() (in module termios), 246
tcflow() (in module termios), 246
tcflush() (in module termios), 246
tcgetattr() (in module termios), 143
tcsetattr() (in module termios), 246
tcsetpgrp() (in module os), 143
tearDown() (TestCase method), 116
tell()
method, 225
aifc method, 396, 397
AU_read method, 399
AU_write method, 399
Chunk method, 402
file method, 15
MultiFile method, 337
Wave_read method, 401
Wave_write method, 401
Telnet (class in telnetlib), 309
telnetlib (standard module), 309
tempdir (data in tempfile), 186
tempfile (standard module), 186
Template (class in pipes), 249
template (data in tempfile), 186
tempnam() (in module os), 146
temporary
file, 186
file name, 186
TemporaryFile() (in module tempfile), 186
termattrs() (in module curses), 169
TERMIOs (standard module), 247
termios (built-in module), 245, 247
termmame () (in module curses), 169
test()
in module cgi, 280
mutex method, 164
testandset () (mutex method), 164
TestCase (class in unittest), 115
TestLoader (class in unittest), 115
testMethodPrefix (TestLoader attribute), 119
tests (data in imghdr), 404
testsRun (TestResult attribute), 118
TestSuite (class in unittest), 115
testzip() (ZipFile method), 233
Textbox (class in curses.textpad), 179
textdomain () (in module gettext), 199
TextTestRunner (class in unittest), 115
Thread (class in threading), 217, 222
thread (built-in module), 215
threading (standard module), 217

threads
  IRIX, 217
  POSIX, 216
tie() (in module fl), 446
tigetflag() (in module curses), 169
tigetnum() (in module curses), 169
tigetstr() (in module curses), 169
time() (in module time), 162
time (built-in module), 159
Time2InternalDate() (in module imaplib), 301
timegm() (in module calendar), 134
timeout() (method), 175
times() (in module os), 148
timezone (data in time), 162
title() (string method), 8
TMP_MAX (data in os), 146
TMPDIR, 186
tmpfile() (in module os), 142
tmpnam() (in module os), 146
tochild (Popen4 attribute), 159
tofile() (array method), 128
togglepause() (CD player method), 444
tok_name (data in token), 427
token
  shlex attribute, 137
  standard module, 427
tokenize() (in module tokenize), 427
tokenize (standard module), 427
tolist()
  array method, 128
  AST method, 420
  xrange method, 10
tomono() (in module audioop), 393
top()
  method, 184
  POP3 method, 299
top_panel() (in module curses.panel), 183
tostereo() (in module audioop), 393
tostring() (array method), 128
totuple() (AST method), 420
touchline() (method), 175
touchwin() (method), 175
tovideo() (in module imageop), 394
toxml() (method), 374
tparm() (in module imageop), 394
trace()
  in module curses, 170
  in module inspect, 52
trace function, 33

traceback
  object, 30, 52
  tracebacklimit (data in sys), 33
TracebackType (data in types), 43
transfercmd() (FTP method), 297
translate()
  in module string, 79
  string method, 8
translate_references() (XMLParser method), 388
translation() (in module gettext), 200
true, 3
truncate() (file method), 15

truth
  value, 3
truth() (in module operator), 45
try
  statement, 17
  ttoob() in module imgfile, 454
  in module rgbimg, 404
tty
  I/O control, 245, 247
tty (standard module), 247
ttynname() (in module os), 143
tuple
  object, 6
tuple() (in module ), 27
tuple2ast() (in module parser), 419
TupleType (data in types), 42
turnoff_sigfpe() (in module fpectl), 39
turnon_sigfpe() (in module fpectl), 39
type
  Boolean, 3
  object, 27
  operations on dictionary, 11
  operations on list, 11
type()
  built-in function, 14, 42
  in module , 27
typeahead() (in module curses), 170
typecode (array attribute), 127
TypeError (exception in exceptions), 19
types
  built-in, 3
  mutable sequence, 10
  operations on integer, 6
  operations on mapping, 11
  operations on mutable sequence, 11
  operations on numeric, 5
  operations on sequence, 7, 11
types (standard module), 14, 27, 41
types_map (data in mimetypes), 345
TupleType (data in types), 42
TZ, 468
tzname (data in time), 162
U
U (data in re), 84
u-LAW, 391, 396, 405, 457
uggettext() (NullTranslations method), 201
uid() (IMAP4 method), 303
uidl() (POP3 method), 300
ulaw2lin() (in module audioop), 393
umask() (in module os), 141
uname() (in module os), 141
UnboundLocalError (exception in exceptions), 19
UnboundMethodType (data in types), 42
unbuffered I/O, 25
unctrl1()
in module curses, 170
in module curses.asci, 182
undoc_header (Cmd attribute), 135
unfreeze_form() (form method), 447
unfreeze_object() (FORMS object method), 449
ungetch()in module curses, 170
in module msvcr, 460
ungetmouse() (in module curses), 170
unhexlify() (in module binascii), 340
unichr() (in module ), 27
UNICODE (data in re), 84
Unicode, 96, 100
database, 100
object, 6
unicode() (in module ), 27
unicodedata (standard module), 100
UnicodeError (exception in exceptions), 19
UnicodeType (data in types), 42
uniform()
in module random, 124
in module whrandom, 125
unittest (standard module), 110
UNIX
file control, 248
I/O control, 248
UnixMailbox (class in mailbox), 346
unknown_charref()
SGMLParser method, 355
XMLParser method, 389
unknown_endtag()
SGMLParser method, 355
XMLParser method, 389
unknown_entityref()
SGMLParser method, 355
XMLParser method, 389
unknown_open()
BaseHandler method, 291
UnknownHandler method, 293
unknown_starttag()
SGMLParser method, 355
XMLParser method, 389
UnknownHandler (class in urllib2), 289
unlink()
method, 374
in module os, 146
unlock() (mutex method), 164
unmimify() (in module mimify), 350
unpack() (in module struct), 88
unpack_array() (Unpacker method), 343
unpack_bytes() (Unpacker method), 343
unpack_double() (Unpacker method), 342
unpack_farray() (Unpacker method), 343
unpack_float() (Unpacker method), 342
unpack_fopaque() (Unpacker method), 343
unpack_fstring() (Unpacker method), 343
unpack_list() (Unpacker method), 343
unpack_opaque() (Unpacker method), 343
unpack_string() (Unpacker method), 343
Unpacker (class in xdrlib), 341
unparsedEntityDecl() (DTDHandler method), 382
UnparsedEntityDeclHandler() (xmlparser method), 360
Unpickler (class in pickle), 56
unqdevice() (in module fl), 446
unquote() (in module urllib), 284
unquote_plus() (in module urllib), 284
unregister() (method), 215
unsubscribe() (IMAP4 method), 303
untouchwin() (method), 175
unused_data (attribute), 231
update()
dictionary method, 11
md5 method, 408
sha method, 408
update_panels() (in module curses.panel), 183
upper()
in module string, 79
string method, 9
uppercase (data in string), 77
URL, 277, 283, 312, 315, 351
parsing, 312
relative, 312
urlcleanup() (in module urllib), 284
urlencode() (in module urllib), 284
URLError (exception in urllib2), 287
urljoin() (in module urlparse), 313
urllib (standard module), 283, 293
urllib2 (standard module), 287
urlopen()in module urllib, 283
in module urllib2, 287
Index 517

URLopener (class in urllib), 285
urlparse() (in module urlparse), 312
urlparse (standard module), 286, 312
urlretrieve() (in module urllib), 284
urlunparse() (in module urlparse), 313
use_env() (in module curses), 170
use_rawinput (Cmd attribute), 135
USER, 164, 296
user
  configuration file, 74
  effective id, 140
  id, 141
  id, setting, 141
user() (POP3 method), 299
user (standard module), 74
UserDict
  class in UserDict, 43
  standard module, 43
UserList
  class in UserList, 43
  standard module, 43
USERNAME, 164
userptr () (method), 184
UserString
  class in UserString, 44
  standard module, 44
UserWarning (exception in exceptions), 20
UTC, 159
utime() (in module os), 146
uu (standard module), 339, 339

V

value
  truth, 3
value (Morsel attribute), 320
value_decode () (BaseCookie method), 320
value_encode () (BaseCookie method), 320
ValueError (exception in exceptions), 19
values() (dictionary method), 11
varray() (in module gl), 452
vars() (in module), 27
VERBOSE (data in re), 85
verbose (data in tabnanny), 428
verify () (SMTP method), 308
verify_request () (in module SocketServer), 315
version
  data in curses, 175
  data in sys, 33
  URLopener attribute, 286
version_info (data in sys), 34
version_string() (BaseHTTPRequestHandler method), 317
vline () (method), 175

W

W_OK (data in os), 144
wait ()
  Condition method, 220
  Event method, 221
  in module os, 148
  Popen4 method, 159
waitpid() (in module os), 149
walk() (in module os.path), 152
warn() (in module warnings), 64
warn_explicit () (in module warnings), 64
Warning (exception in exceptions), 20
warning() (ErrorHandler method), 383
warnings, 62
warnings (standard module), 62
wasSuccessful () (TestResult method), 118
wave (standard module), 399
WeakKeyDictionary (class in weakref), 36
weakref (extension module), 35
WeakValueDictionary (class in weakref), 36
webbrowser (standard module), 275
weekday() (in module calendar), 133
weibullvariate () (in module random), 124
WEXITSTATUS () (in module os), 149
wfile (BaseHTTPRequestHandler attribute), 316
what ()
  in module imghdr, 404
  in module sndhdr, 405
whatdr () (in module sndhdr), 405
whichdb () (in module whichdb), 227
whichdb (standard module), 227
while
  statement, 3
whitespace
  data in string, 78
  shlex attribute, 137
wrandom (standard module), 125
wrandseed attribute, 137
wrandseed () (in module random), 123
WIFEXITED () (in module os), 149
WIFSIGNALED () (in module os), 149
WIFSTOPPED () (in module os), 149
window () (method), 184
Windows ini file, 129
WindowsError (exception in exceptions), 19
WinSock, 215
winsound (built-in module), 465
winver (data in os), 149
WNOHANG (data in os), 149
wordchars (shlex attribute), 137
World-Wide Web, 275, 283, 312, 351
wrapper() (in module curses.wrapper), 180
writable() (dispatcher method), 324
write()
    method, 225
        array method, 128
        audio device method, 458
        ConfigParser method, 131
        file method, 15
        in module imgfile, 454
        in module os, 144
        InteractiveConsole method, 68
        StreamWriter method, 98
        Telnet method, 311
        ZipFile method, 233
write_byte() (method), 225
write_history_file() (in module readline), 235
writeframes()
    aifc method, 397
    AU_write method, 399
    Wave_write method, 401
writeframesraw()
    aifc method, 397
    AU_write method, 399
    Wave_write method, 401
writelines()
    file method, 15
    StreamWriter method, 98
writepy() (PyZipFile method), 233
writer (formatter attribute), 328
writesamps() (audio port method), 441
writestr() (ZipFile method), 233
writexml() (method), 374
WrongDocumentErr (exception in xml.dom), 371
WSTOPSIG() (in module os), 149
WTERMSIG() (in module os), 149
WWW, 275, 283, 312, 351
    server, 277, 315

X

X (data in re), 84
X_OK (data in os), 144
xatom() (IMAP4 method), 303
XDR, 54, 341
xdrlib (standard module), 341
xgtitle() (NNTPDataError method), 306
xhdr() (NNTPDataError method), 306
XML, 387
    namespaces, 390
xml.dom (standard module), 363
xml.dom.minidom (standard module), 373
xml.dom.pulldom (standard module), 377
xml.parsers.expat (standard module), 357
xml.sax (standard module), 377
xml.sax.handler (standard module), 379
xml.sax.saxutils (standard module), 383
xml.sax.xmlreader (standard module), 383
XmlDeclHandler() (xmlparser method), 359
XMLFilterBase (class in xml.sax.saxutils), 383
XMLGenerator (class in xml.sax.saxutils), 383
xmlib (standard module), 387
XMLParser (class in xmlib), 387
XMLParserType (data in xml.parsers.expat), 357
XMLReader (class in xml.sax.xmlreader), 383
xor() (in module operator), 45
xover() (NNTPDataError method), 306
xpath() (NNTPDataError method), 306

Y

Y2K, 159
Year 2000, 159
Year 2038, 159
yiq_to_rgb() (in module colorsys), 403

Z

ZeroDivisionError (exception in exceptions), 20
zfill() (in module string), 80
zip() (in module), 28
ZIP_DEFLATED (data in zipfile), 232
ZIP_STORED (data in zipfile), 232
ZipFile (class in zipfile), 232
zipfile (standard module), 232
ZipInfo (class in zipfile), 232
zlib (built-in module), 229